The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

To renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

MAR 17 1992

APR 28 1992

APR 03 1996

JAN 06 1997

L161—O-1096

Math

Report No. UIUCDCS-R-77-907

UILU-ENG 77 1764

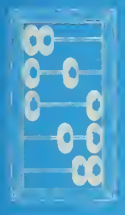
MULTIPROCESSOR FOR STRING MANIPULATION

by

Wing Kai Cheng

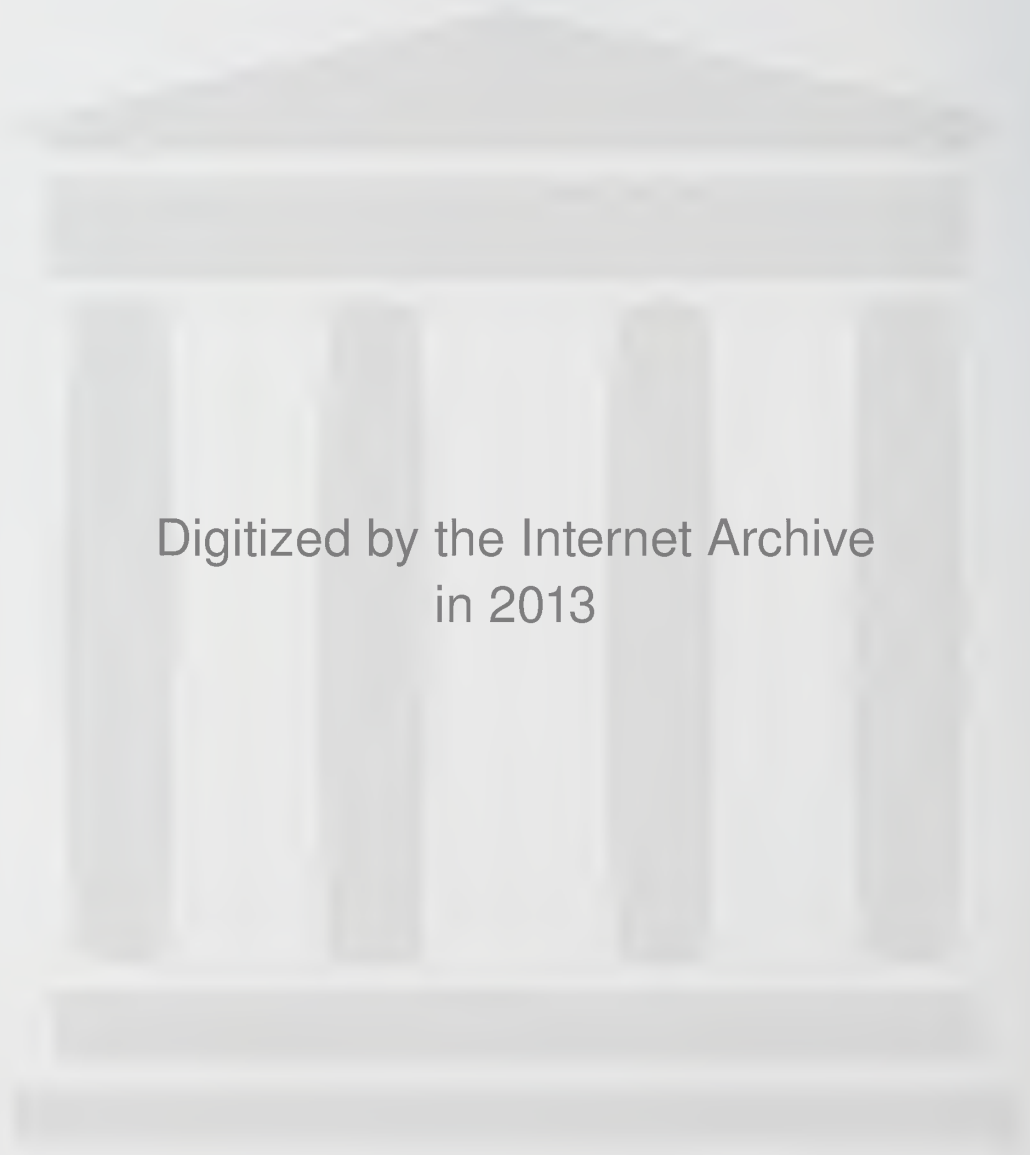
October 1977

NSF-OCA-MCS73-07980-000029



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the
MAR 2 1 1978
University of Illinois
at Urbana-Champaign


Digitized by the Internet Archive
in 2013

http://archive.org/details/multiprocessorfo907chen

Report No. UIUCDCS-R-77-907

MULTIPROCESSOR FOR STRING MANIPULATION*

by

Wing Kai Cheng

October 1977

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF-MCS73-07980 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, October 1977.

ACKNOWLEDGEMENT

I would like to thank my advisor Professor David J. Kuck for his advice, guidance, incisive questions, encouragement, and general concern.

Many others also deserve much thanks--a partial list includes Professor R. P. Futrelle, my friends, and my parents.

TABLE OF CONTENTS

PAGE

1. INTRODUCTION.....1

2. AN OVERVIEW OF SNOBOL.....4

2.1 Introduction.....4

2.2 Syntax and Semantics.....5

2.3 Serial Execution.....11

2.3.1 Program Execution.....11

2.3.2 Statement Exectuion.....11

2.3.3 Serial Pattern Matching.....13

2.4 Speedup.....16

2.4.1 Speedup of the Basic Operations.....16

2.4.2 Speedup at Statement Level.....17

2.4.3 Speedup in the Program Level.....17

2.4.4 IF Tree.....18

3. PATTERNS AND PATTERN MATCHING PROCESSOR.....22

3.1 Introduction.....22

3.2 Algebra of Pattern matching.....22

3.2.1 Deviation from SNOBOL4.....28

3.3 Time Bound for Pattern Matching Algorithm in SNOBOL4.....29

3.4 Pattern Matching Network.....31

3.4.1 Stage I.....37
3.4.1.1 Comparison of the Methods.....40
3.4.2 Design of Stage I.....41
3.4.2.1 Speed.....50
3.4.3 Stage II.....50
3.4.4 Stage III.....50
3.4.5 Stage IV.....51
3.4.6 Stage V.....51
3.4.7 Stage VIa.....51
3.4.8 Stage VIb.....52
3.5 Gate, Time, and Gate*time.....52
4. MACHINE DESIGN.....66
4.1 Overall Organization.....66
4.2 Garbage Collection.....67
4.3 IF Tree Processor.....69
4.4 Memory.....70
4.5 Program Memory.....72
4.6 Data Memory.....72
4.7 Instruction Set.....77
5. EMPIRICAL RESULTS.....81
5.1 Introduction.....81
5.2 Static Frequency Count.....82

5.3 Dynamic Statistics.....87
6. SYSTEM PARAMETERS.....88
6.1 Introduction.....88
6.2 Descriptor Size.....88
6.3 Descriptor Format.....88
6.4 Character Size.....89
6.5 ALU Processor.....90
6.6 IF Free Processor.....91
6.7 Program Memory.....91
6.8 Data Memory.....92
6.9 PMI.....93
6.10 Expected Speedup.....94
7. CONCLUSION.....96
LIST OF REFERENCES.....97

1. INTRODUCTION

In the past decade, there has been significant improvement in computer speed as a result of faster and faster gates, and computer organization which exploits parallel processing. This trend is expected to continue, with major improvement to come from computer organization, as switching speed is approaching its theoretical limit, and the declining cost and the progress of LSI technology are making the exploitation of parallelism economically and physically possible. It is this computer organization (machine and software) aspect which this thesis is concerned with.

Many proposals and attempts have been made to speed-up the execution of numerical problems on computers, and each has achieved some form of success. In some cases, actual machines have been built or are being built: some examples of which are ILLIAC IV, CRAY I, CDC 7700, CDC STAR, TI ASC, etc.

On the other hand, computer organization for

non-numerical computing is still in its infancy, although there are probably more non-numerical problems than numerical ones in this world. The following result is probably familiar to many people [GIM76]: To multiply two 3 digits numbers, it takes the computer 6 microseconds as compared to about 60 seconds by human-- the computer is therefore 10**7 times faster than human. To search for a ten character string in a 3000 characters text takes a typical computer 6 milliseconds, whereas it takes a human being 60 seconds-- thus the machine is 10**4 times faster than human. Comparing the two situations, one may conclude that the machine is about 10**3 times faster with numerical problems than it is with non-numerical problems. The obvious reason for this inefficiency for non-numerical problems is that non-numerical problems and languages contain facilities and concepts so far removed from the conventional machines that elaborate software is required to bridge the gap. Furthermore, parallelism and functional requirements are not as well understood and exploited in machines for these non-numerical problems as they are for the numerical counterparts.

With the increasing usage of computers for non-numerical data processing, it has become more important to process this kind of data more efficiently and faster. The primary objective of this research is to study the requirements and peculiarities of string processing, and to propose a machine organization that is suitable for string processing in terms of speed and ease of supporting software features of their languages. A well known string processing language called SNOBOL[POA71] will be used as a representative string processing language for our study.

We start off with Chapter 2 as a synopsis of the basic features of SNOBOL, and an introduction to our approach of speeding up SNOBOL. Algebraic properties of patterns and the design of a pattern matching network are presented in chapter 3. The machine organization and its instruction set are described in Chapter 4. Some empirical results of SNOBOL programs are given in Chapter 5. Chapter 6 discusses the system parameters and gives an estimation of the speedup achievable. Conclusions and suggestions for further research can be found in Chapter 7.

2. AN OVERVIEW OF SNOBOL

2.1 Introduction

SNOBOL is mainly a language for string manipulation, containing many features not readily found in other programming languages. This language proves to be a useful tool in areas such as compilation, machine simulations, symbolic mathematics, text preparation, natural language translation, linguistics, music analysis, information retrieval, artificial intelligence research, and the like. The fact that it is implemented in several different computers (including IBM System /360 and 370, UNIVAC 1108, GE635, CDC 3600, CDC 6000 series, PDP10, Sigma 5/6/7, Atlas 2, and RCA Spectra Series) probably reflects the wide acceptance by the users.

The basic operations in SNOBOL, in addition to the usual arithmetic and assignments, include concatenation, alternation, and pattern matching. The basic data element of SNOBOL is the string, that is, a string of characters. SNOBOL also provides numerical capabilities with both integers and real numbers, and automatic conversion

capability between string and numerical values. Integer is more often used than real number because most numerical operations involve character counting.

Structured data types include those of arrays, which can be accessed through index, and tables, which can be accessed through variables.

2.2 Syntax and Semantics

A brief description of the SNOBOL syntax and semantics is provided in this section in order to point out some of the peculiarities and properties of SNOBOL and highlight features we propose for speedup. A more detailed and complete description of SNOBOL can be found in [POA71,GRI72].

The syntax of SNOBOL will be defined by the traditional Backus-Naur form [BAC59,NAU63], where syntactic constructs are denoted by English words in lower case letters. Sequences of constructs separated by the meta-symbols (and) imply their repetition zero or more

times. In addition, we choose to use '::=' to separate left part of the production from the possible right parts. The notation [x] will be used to indicate that the term x is optional.

PRODUCTION RULES	COMMENT
program::=([label] statement) END [character-string]	
label::=character string	
statement::=[rule]	no goto
::=rule : goto	unconditional goto
::=rule : S goto	S goto
::=rule : S goto1 F goto2	S-F goto
::=rule : F goto	F goto
::=rule : F goto1 S goto2	F-S goto
goto::=(expr)	normal goto
::= <expr>	direct goto
rule::=[expr]	expression evaluation
::=expr =	null assignment
::=expr1 = expr2	assignment
::=expr1 expr2	pattern match
::=expr1 expr2 =	null replacement
::=expr1 expr2 = expr3	string replacement
expr::='character-string'	literal
::=number	number
::=identifier	identifier
::=operator expr	unary operation
::=expr1 operator expr2	binary operation
::=identifier ((expr))	function call
::=identifier <(expr)>	array indexing

Table 2.2-1 Simplified SNOBOL Syntax

A SNUBOL program consists of a sequence of statements which are executed sequentially unless control is transferred elsewhere. A statement, as shown in Table 2.2-1, contains up to two parts: rule and goto. The execution of a statement merely determines which statement is to be evaluated next; in other words, the statement itself carries with it no value and does not succeed or fail. Within a statement, rule is evaluated first, and its success or failure is used to influence the goto part to determine the subsequent statement to be evaluated. If the statement is without a goto, the subsequent statement is evaluated; however, if the statement is with a conditional goto, control is transferred to the statement indicated by the evaluation of the goto. S goto or F goto is evaluated when the rule signals success or failure respectively. If the rule signals success (failure) and the statement is without S goto (F goto), control is simply transferred to the following statement in the program.

The three main classes of statements are (1)assignment, (2)pattern matching, and (3)replacement,

depending on the rule part.

As shown in Table 2.2-1, assignment has the form

subject = object

Pattern matching takes the general form

subject pattern

whereby during execution, subject is inspected for the occurrence of pattern.

The replacement statement is a combination of pattern matching and assignment, in which the matched string is replaced by the string in the object field. The form is

subject pattern = object

The result of the execution of the rule is either a success or failure. If the evaluation of any one of expr, expr1,expr2, or expr3 should fail, then no further parts of the rule are evaluated, and the whole rule fails. More specifically, if the rule is an expression (expr) evaluation, null assignment, or assignment, the rule succeeds if none of the expr fails.

In pattern matching, expr1 and then expr2 are evaluated and coerced. Expr1 is then used as the subject or text, and expr2 is used as the pattern. Pattern matching either succeeds or fails.

In the case of the pattern match replacement, expr1 and expr2 are evaluated as in pattern match. If the match succeeds, expr3 replaces the matched substring through the use of concatenations. If expr3 is missing from the rule, expr3 is assumed to be a null string.

The evaluation of an expression may either succeed or fail. If evaluation succeeds, the result may either be returned by name (when the result of the evaluation is a location) or returned by value (when the result is a storable value). If the evaluation fails, it is known as fail return. If a location is returned by name when a value is needed in an operation, the operation may retrieve the value associated with the location; this process of retrieval for value when a location is returned is known as coercion.

2.3 Serial Execution

The order of execution in a serial SNOBOL program is presented in this section.

2.3.1 Program Execution

As noted earlier, statements in a SNOBOL program are executed in a sequential order unless control is transferred elsewhere by a goto. Transfer of control can be either unconditional or conditional, as in other programming languages.

2.3.2 Statement Execution

A SNOBOL statement, having the general form

subject pattern = object goto is evaluated in a sequential order as follows [POA71]:

1. The subject is evaluated first. If the evaluation fails, the statement fails, goto is evaluated, and the other components of the statement are not evaluated. If no failure goto is specified,

control is passed to the next statement.

2. The pattern is then evaluated. If the evaluation fails, goto is evaluated as in the case of subject failure in (1).

3. The pattern match is performed next. If the match fails, the statement fails, conditional value assignment is not performed, the replacement is skipped, and the goto is processed. Immediate value assignments may take place before the pattern match fails. If pattern matching succeeds, conditional value assignment is performed for those components that matched.

4. The object is evaluated. If evaluation fails, the statement fails, no replacement is performed, and the goto is processed.

5. The replacement is performed.

6. The goto is processed. The three possibilities are:

    (a) If the statement succeeds, only an unconditional or success goto in the statement is processed.

(b) If the statement fails, only an unconditional goto or failure goto in the statement is evaluated.

(c) Transfer is made to an evaluated goto, if there is one.

If none of the three conditions above is met, control is transferred to the next statement.

2.3.3 Serial Pattern Matching

Pattern matching can be a very time consuming process, as illustrated by the following example.

Consider a pattern of the form

ST = ('B'!'BRE') ('E'!'A') 'D'

which has three subsequents to the right of the equality sign, where the first and second has two alternates each. '!' is used to denote alternation, the spaces between ('B'!'BRE') and ('E'!'A') and that between ('E'!'A') and 'D' are concatenation operators.

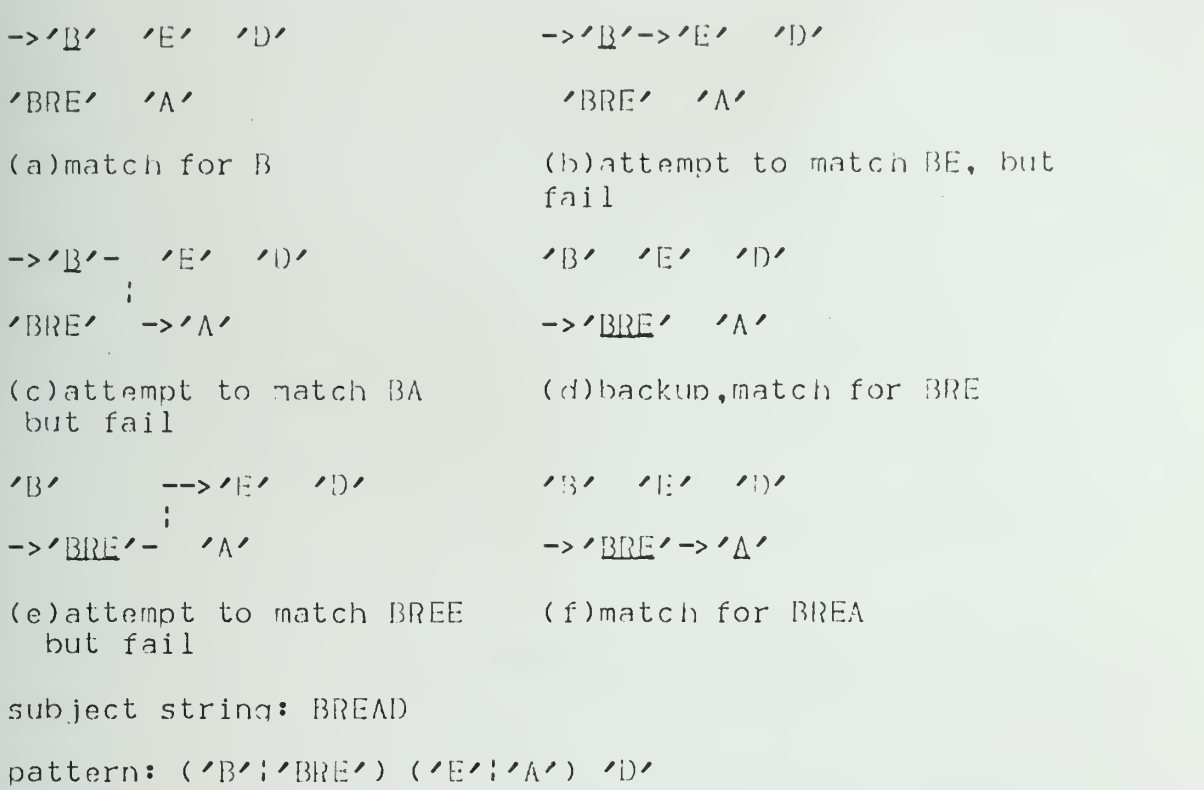
The process of pattern matching for the pattern SF, which matches any of the string in the set (BED,BAD,BREED,BREAD) is shown in Fig. 2.3.3-1. The matching algorithm for SNOBOL4 [POA71,GRI72] may be summarized as follows:

1. Matching starts with the first alternate of the first subsequent.

2. When a component matches, the next subsequent to the right is attempted; otherwise, goto (4).

3. If a match can be obtained for a series of components leading from the left through the right, the pattern match is successful.

4. If a component does not match, an alternate within this subsequent is tried. If no alternate exists, the matching process backs up, rejecting the previously matched component and seeks an alternate for it.



subject string: BREAD

pattern: ('B'|'BRE') ('E'|'A') 'D'

Fig. 2.3.3-1 Snapshots of a pattern matching process in SNOBOL4

2.4 Speedup

The speedup proposed will be a combination of special hardware and algorithms. Usually, for a given programming language, one can expect that parallelism may be exploited in three different levels: (1) In basic numerical and non-numerical operations, (2) statement level, and (3) program level. All these methods will be explored in more detail in the following sections.

2.4.1 Speedup of the Basic Operations

Speedup of the basic numerical operations, such as addition of two numbers or multiplication of two numbers, has been investigated by many people, such as [DAD65,MSO61].

In the next chapter, we shall propose a way to speedup pattern matching. Speedup of concatenation and alternation can be expected to come mainly from memory bandwidth.

2.4.2 Speedup at Statement Level

To speedup at the statement level is to execute as many of the components of a statement in parallel as possible. For a numerical statement, it is to evaluate the numerical expression through multiprocessing [MUR71,KRA72,KUC72].

For example, to evaluate

$$a=b+c+d+e,$$

one needs a minimum time of two adder delays, using three adders. In general, to evaluate an expression consisting of n terms requires $\lceil \log_2 n \rceil$ time delay and n-1 processors. An analog to aritmetic expression in string processing is to perform a pattern matching involving concatenations and alternations; speedup of this is discussed in a later chapter.

2.4.3 Speedup in the Program Level

Speedup in the program level involves the execution of more than one statements at one time. Transformations

of the program are required to remove data dependency of the statements [IUW76]. An example of these transformations is forward substitution. The statements

```
a = c d
b = a c e
f = b g
```

can be transformed into

```
a = c d
b = c d c e
f = c d c e g
```

after forward substitution. Interdependencies are removed in the transformed statements: thus, all three statements are executable in parallel. Care must be taken to insure that the variables refer to the same value before forward substitution is performed, since this process is complicated by the fact that SNOBOL allows input association, immediate value assignment, and conditional value assignment.

2.4.4 IF Free

In SNOBOL, let us refer to the statements with

S-goto, F-goto, S-F-goto as decision statements, and collectively call the other type of statements the assignment statements.

In a program, assignment statements are interspersed with decision statements. The parts of a program where the ratio of decision statements to assignment statement operations is larger than an experimentally determined threshold are called IF trees. A decision tree is formed when the assignment statements are removed from the IF tree.

To speedup the execution of a program having a significant number of decision statements, an IF tree processor may be used. The idea behind the IF tree processor [DAV72b] is to execute all possible paths and then pick the right results. In order to do this, the following steps have to be performed at compile time:

1. A new assignment statement, the left hand side of which is a logical variable is added to each decision statement. These new statements will store the results (success or failure) of the decision

statements. We will later refer to the values of these logical variables as SF values.

2. Move all the decision statements plus the statements they depend on, to a position ahead of the decision tree; call these IFBS. Move the rest of the statements below the tree. Note that the statements have to be tagged before the movement so as to identify their positions in the IF tree.

3. Map the nodes from the decision tree to the processor.

During the execution phase,

1. All statements ahead of the tree are evaluated as fast as possible.

2. The SF values that are needed for the decision are given to the IF tree processor.

3. Evaluate the tree in the IF tree processor and find the proper path.

4. Based on the path obtained in Step 3, the proper results from the IFBS are selected, and the statements which had been moved down in compile time

and which are in the selected path are evaluated.

The rest of the algorithms for IF-tree and its hardware design are similar to that proposed by Davis [DAV72b].

3. PATTERNS AND PATTERN MATCHING PROCESSOR

3.1 Introduction

One of the main features and a time consuming process of SNOBOL is pattern matching. In this chapter, we will discuss the algebra of pattern matching, some pattern matching schemes, and the design of a pattern matching processor.

3.2 Algebra of Pattern Matching

Consider the set of characters, consisting of letters, digits, punctuation symbols, null character, NONE(character which does not match anything), and ALL (character which matches any character). We define a string as a sequence of one or more characters. Clearly, a sequence of null characters is called a null string.

Let us denote the length of a string s by |s|. By definition, the length of a null string is considered to be zero.

If C is a character set and aϵC, then a%S or S%a is again a string, of length |s|+1, where the symbol '%' is used to denote the concatenation operator (instead of a space as in SNOBOL) for the sake of clarity, except in places where SNOBOL examples are used. With these definitions agreed upon, we can try to summarize the properties of pattern matching that we will use later. Previous work in this area of the formalization of pattern matching had been done by Gimpel[GIM73], and Stewart [STE74].

Pattern matching can be loosely defined as the process of examining the occurrences of a set of substrings called patterns within a longer string called subject string. The set of patterns can be expressed with strings, concatenation operators, and alternation operators. A pattern without any concatenation or alternation operator (that is, consisting of a simple string) is referred to as a simple pattern.

Def Pattern matching is a function M(s,p,c) where s is a string called the subject string or text, p is a string

called the pattern, and c, called the pre-cursor position, is a pointer to s, taking on the values in the range 1,...,|s|-|p|+1. The value of M(s,p,c) is an integer, called the post-cursor position, pointing to the text where the match begins. We can extend the definition a little further to define a pattern matching which finds all the matches as

$$AM(p,s,c)=\cup M(p,s,c)$$

for all c, where $\cup$ denotes set union.

If the post-cursor takes on a value outside the given range of 1,...|s|-|p|+1, the match is considered to have failed. The patterns FAIL and NULL in SNOBOL can be formally defined as follows:

Def FAIL is a pattern defined by

$$M(s,FAIL,c) = 0$$

Note that M(s,FAIL,0)=0.

Def NULL is a pattern defined by

$$M(s,NULL,c)=c$$

Example

If p=CAB and s=ABACABA, and c=0, then M(s,p,c)=4.

One can think of an inverse of a pattern p, as a pattern which doesn't match p. As an example, the pattern in SNOBOL which matches any character except the character 'A,' written in SNOBOL as

'A' ABORT

In general, a pattern which matches B but not A is written

PAT = A ABORT | B

Two patterns P1 and P2 are equal iff

M(s,P1,c)=M(s,P2,c) for all s and c, or

AM(s,P1,c)=AM(s,P2,c).

Def The concatenation of P and Q, represented here as P%Q, where P and Q are strings, is defined as

$$M(s,P\%Q,c)=M(s,P,c)\cap[M(s,Q,c)-|P|]$$

and in the case for all matches

$$AM(s,P\%Q,c)=AM(s,P,c)\cap[AM(s,Q,c)-|p|].$$

An alternation is an operation which assign more than one string to a variable; it can be defined in the following manner.

Def The alternation of P and Q, represented as P|Q, is defined as M(s,P|Q,c)=M(s,P,c) U M(s,Q,c) and

AM(s,P|Q,c)=AM(s,P,c) U AM (s,Q,c), where U denotes the set union.

It is easy to show that the following properties hold under the operations of concatenation and alternation give P and Q are patterns; thus, the properties are listed without proof.

Let P be the set of patterns

P1. P is closed under alternation. If $p \in P$ and $q \in P$, then $p|q \in P$ and $q|p \in P$.

P2. Commutative law holds under alternation. If $p,q \in P$ then $p|q=q|p$.

P3. Associative law holds under alternation, i.e. if $p,q,r \in P$ then

$$(p|q)|r=p|(q|r).$$

P4. There exists an identity element FAIL in P such that

$$p|FAIL=p.$$

P5. P is closed under concatenation. If $p,q \in P$, then

$$p\%q \in P \text{ and}$$

$$q\%p \in P.$$

P6. Associative law holds under concatenation

$$(p\%q)\%r=p\%(q\%r)$$

P7. Distributive law holds

$$p\%(q|r)=(p\%q)|(p\%r) \text{ and}$$

$$(q|r)\%p=(q\%p)|(r\%p)$$

P8. There exists an inverse $-p$ such that

$$p|-p=FAIL$$

P9. $p\%NULL=NULL\%p=p$

P10. Any pattern formed by concatenations and alternations.

$$(P_{11}|P_{12}|\ldots|P_{1m_1})\%(P_{21}|P_{22}|\ldots|P_{2m_2})\%\ldots\%(P_{n1}|\ldots|P_{nm_n})$$
$$=P_{11}\%P_{21}\%\ldots\%P_{n1}|P_{12}\%\ldots\%P_{n1}|\ldots|P_{1m_1}\%\ldots\%P_{nm_n}$$

As a side remark, patterns, under the operations of alternation and concatenation, form a ring with unit element[HER64] as a result of P1 through P10.

3.2.1 Deviation from SNOBOL4

Readers familiar with SNOBOL4 would probably notice that P2, the commutativity of the alternation operations (p|q=q|p) is not strictly valid in SNOBOL4;in addition, pattern matching in SNOBOL4 returns only the first match, rather than all the matches.

The non-commutativity in SNOBOL4 is actually imposed by the fact that SNOBOL4 systems are implemented on serial machines, where matching is done serially. In a serial implementation, the users would arrange the alternates in the order of the likelihood of successfully matching, to

increase the probability of reducing the execution time. With the use of parallel processors, the lifting of the restriction on commutativity would be an advantage. A special mode can be implemented to force the execution of alternation in a sequential manner as defined in SNOBOL4 if the order is important.

The fact that only the first match is sought in SNOBOL4 can also be regarded as a victim of the serial implementation. Again, in our proposed machine, the programmer can decide whether he wants only the first match or all of the matches. Hardware is shown in the next section which could detect the first match.

3.3 Time Bound for Pattern Matching Algorithm in SNOBOL4

Pattern matching in SNOBOL4 is done in almost an ad hoc manner. The algorithm for pattern matching used in SNOBOL4, given in Section 2.3.3, can easily be implemented using stacks. Due to the unnecessary backtracking of the algorithm, the time required to match two simple strings may be as much as o(pq), as in the case for matching the

pattern A**pB and the subject string A**qB (where o indicates in the order of, and A**n denotes a string of n characters of A). With a slightly more complex pattern, such as one with m alternates it would require as much as mpq comparisons. When concatenation is introduced, such as

$$p=(A1|A2|...|Am)\ (B1|B2|...|Bn)$$

assuming |Ai|=|Bj|=k, and the length of the subject string is s, the time increases to 2kmns.

As pattern matching is the central feature and a very time consuming process, it is of paramount importance to speed it up.

Faster serial pattern matching algorithms for one pattern and one subject string do exist [KNU74], requiring processing time o(m+n) for a subject string of length n and a pattern of length m.

An algorithm for pattern with alternates also exists[AHU75], requiring processing time proportional to o(n+sum of the length of the alternates), where n is the

length of the subject string. Both of the two faster algorithms require some preprocessing. The time for the two faster algorithms quoted here includes the time needed for the pre-processing.

The main weakness of these two algorithms are that they both require pre-processing, and storage for tables generated during pre-processing is required. In the next sections, we will present a scheme which does not require any pre-processing.

3.4 Pattern Matching Network

In general, a pattern P in SNOBOL can be expressed as

$$P = (A_{11}|A_{12}|\ldots|A_{m1})\ (A_{21}|A_{22}|\ldots|A_{2m_2})\ldots(A_{n1}|\ldots|A_{nm_n})$$

where each Aij is a string, '|' denotes alternation, and the space between ')' and '(' denotes concatenation. One may view the statement as analogous to a product of sum. If fully expanded, the above equation is

$$P = A_{11}A_{21}\ldots A_{n1}\,|\,A_{12}A_{21}\ldots A_{n1}\,|\ldots|\,A_{1m_1}A_{2m_2}\ldots A_{nm_n}$$

Given a subject string T such that

$$T=t(1)t(2)\ldots t(s)$$

let us define the pattern matching problem

$$T \; P$$

as a problem to find post-cursor positions for some c or all c such that $1 \leq c \leq s$. We shall refer to a machine which can perform this task as a pattern matching network (PMN), as shown in Fig. 3.4-1. A sub-machine of this network, M(T,A), which produces a bit string or vector of 0's and 1's

$$m(1)\ldots m(s-k+1)$$

called the post-cursor-position-bit-vector for the input text $t(1)\ldots t(n)$ and one simple pattern $a(1)\ldots a(k)$ as input is called pattern matcher(PM), with the condition that $m(i)=1$ whenever $t(i)\ldots t(i+k-1)=a(1)\ldots a(k)$ for some i, $1 \leq i+k-1 \leq n$.

To illustrate how the machine in Fig.3.4-1 works, let us limit the pattern to the simplified case below, with only a limited number concatenations:

$$P = (A_1 | A_2 | \ldots | A_m)\,(B_1 | B_2 | \ldots | B_n) \qquad \text{Eq. 3.4.1}$$
$$= A_1B_1 | A_1B_2 | \ldots | A_mB_n$$

By definition and using the properties of patterns in Section 3.2

$$P = [\{M(S, B_1, C) - |A_1|\} \cap M(S, A_1, C)]$$
$$\cup [\{M(S, B_2, C) - |A_1|\} \cap M(S, A_1, C)]$$
$$\cup \ldots$$
$$\vdots$$
$$\cup [\{M(S, B_n, C) - |A_m|\} \cap M(S, A_m, C)]$$

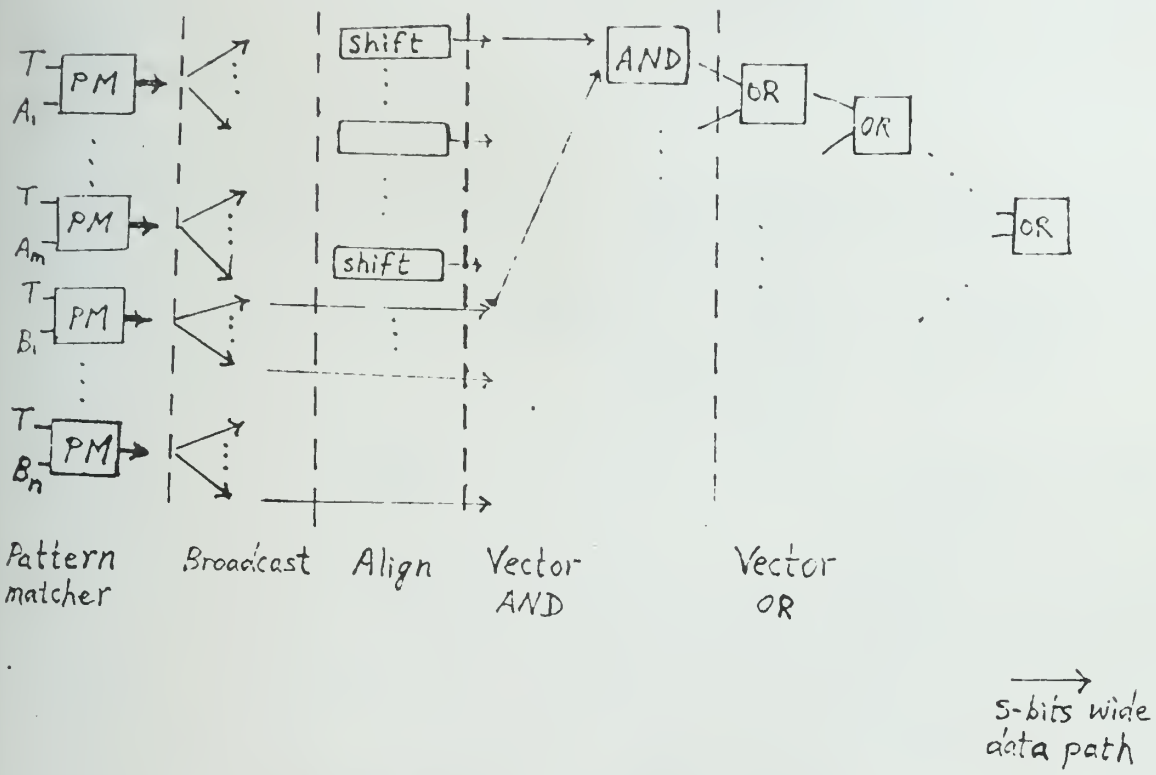
The last equation above suggests that the pattern matching of a complex pattern (Eq 3.4.1) can be broken up (1) into simple pattern matching such as M(s,A1,c), and M(s,B2,c), etc., where each A1,...,Am,B1,...,Bn represents simple patterns, and (2) appropriate subtractions, and union and intersection operations. Step 1 can be solved by using PM where each PM takes a subject string and a simple pattern as inputs, as shown in Stage 1 of Fig.3.4-1. Since the output of the PM are bit vectors,

the subtraction, intersection and union may be replaced by shifting, ANDing, and ORing respectively as shown in Stages III through STAGE IV of Fig.3.4-1. An example connection for a particular pattern is given in Fig.3.4-2.

Note that in an effort to save gates and storage, we do not design the PMN so that it works with the expanded form

A1B1|...|AmBn

as inputs, where we use the notation AiBj to represent the concatenation of Ai and Bj. The output of Stage I are strings of bits, and the rest of the stages work with these strings of bits instead of strings of character or indices, resulting in additional savings in gates and storage.



Broadcast--broadcast each output vector M(T,Ai) to n places, and each M(T,Bi) to m places.
Align--shift each M(T,Ai) that is to be ANDed with M(T,Bj) in the next stage |Bj|-|Ai| positions to the right if |Bj|>|Ai|; else to the left.
Vector AND--vector AND M(T,Ai) and M(T,Bj) for all i and j.
Vector OR--OR the resulting vectors of the previous stage to form one vector.

Fig. 3.4-1 The different stages of PMN

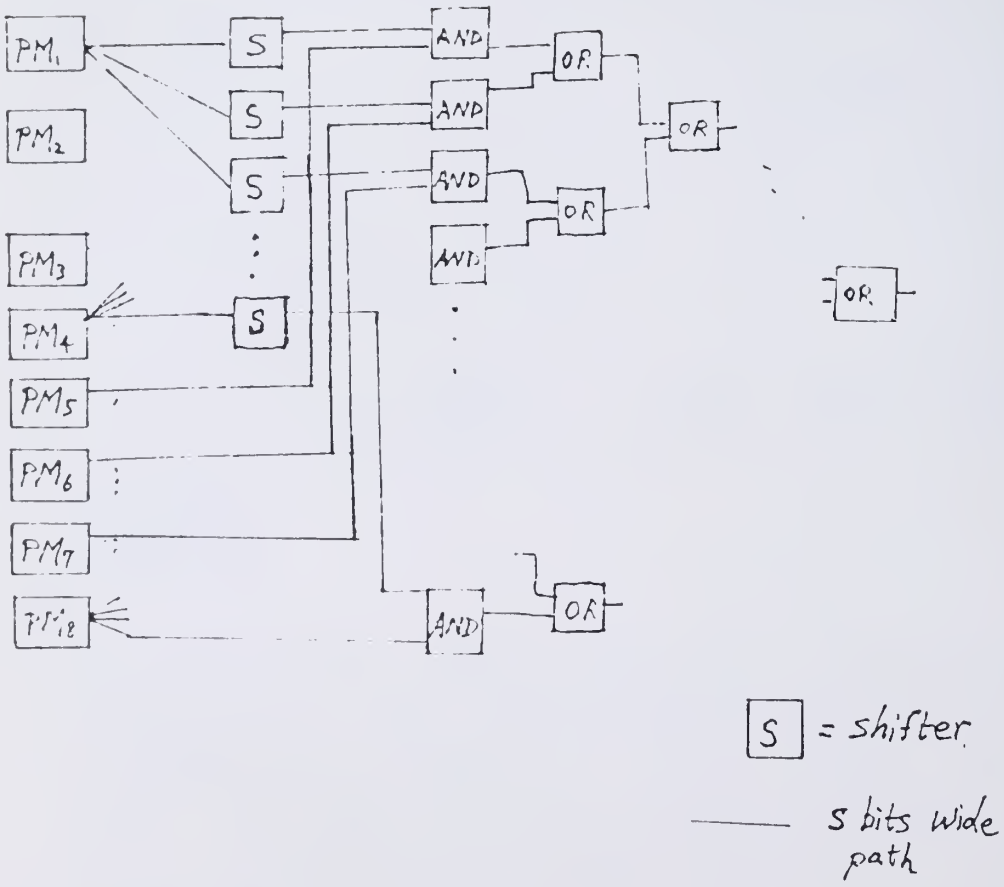


Fig. 3.4-3 Connection for pattern (A1:A2:A3:A4) (B1:B2:B3:B4)

3.4.1 Stage I

Three different ways to implement Stage I are considered:

1. Maximum Parallel(MP)

2. Pattern Serial (PS)

3. Serial using Knuth,Morris, and Pratt's algorithm (SKMP)

Method 1 (MP)

This method involves comparing the pattern in every possible position in relation to the subject string simultaneously. This is the most parallel way conceivable.

Conceptually, this method involves comparing every character of the pattern with every character of the subject string and producing a bit map as shown in Fig.3.4.1-1. The next step is to AND all these bits in groups as outlined by the ellipses.

The time bound for this is asymptotically equal to log2 s, where s is the length of the subject string.

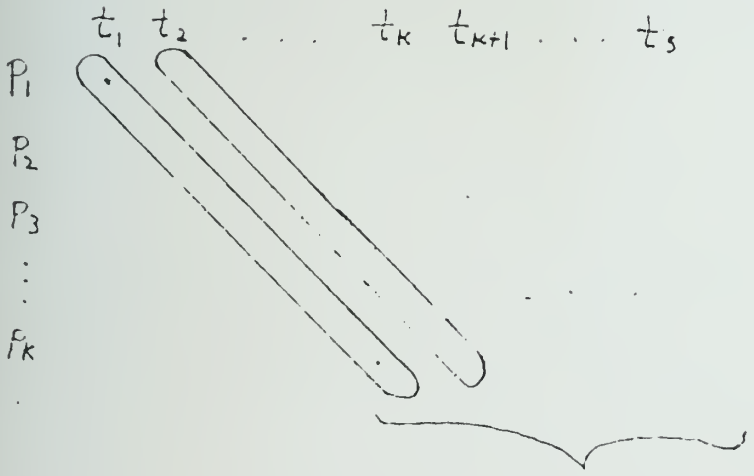


Fig. 3.4.1-1 Maximal parallel way of matching

Method 2 (PS)

This is almost like Method 1 except that only one character of the pattern is compared with the whole subject string at a time or section of the subject string at a time. One may think of this as Method 1 compressed

in space and expanded (iterated) in time in the 'pattern' dimension.

The speed of this method is very close to that of Method 1, especially for short pattern. Using this method, one may take advantage of the fact that in the real world, the pattern is usually short.

Method 3 (SKMP)

This is an implementation of the method of [KNU74,AHO75]. The time required is proportional to s plus preprocessing time.

3.4.1.1 Comparison of the Methods

Among the three methods, MP represents the fastest method with the highest in gate requirement. On the other extreme, SKMP is the slowest and requiring the least number of gates. Method 2, PS, lies between MP and SKMP in terms of gates and time.

To further compare the merits of these three methods, we calculated the values of gate*time for all these three methods. The results show that gate*time remains approximately constant for all these methods; thus, we are unable to select one of the above methods and incorporate it into our design by basing our decision solely on the gate*time criteria.

Eventually, PS,Method 2, was selected based on the following reasons: (1) the performance is very close to that of Method 1 (MP), especially for short patterns, (2)preprocessing is not necessary, and (3) Method 1 is very rigid: It can only process strings of a fixed length; strings of length greater than it is designed for requires additional messy work. One the other hand, Method 2 can process strings of any length with equal ease. Method 2 can be regarded as the golden mean between the extremes. The design is given in the following sections.

3.4.2 Design of Stage I

The function of the PM, Fig.3.4-2, is such that given

the character string inputs p(1)...p(k) and t(1)...t(n), the PM can produce b(1)...b(n). The bit vector b(1)...b(n) is called the post cursor position bit vector, whereby b(i)=1 if p(1)...p(k)=t(i)...t(i+k-1) and 1<=i<i+k-1<=n.

The machine which performs the function of PM of Fig.3.4-3 is shown in Fig. 3.4.2-1, and controlled by Algorithm 3.4.2-1. The subject string is loaded into the X registers X(1)...X(s), and one character of the pattern is placed in register P.

During each minor cycle, the character in P is compared with each character in the X registers. The result is ANDed with the content of D and restored in D, where each D register holds a bit. The characters in the X registers are shifted left and the next character of the subject string is loaded into X(s). The next character in the pattern is loaded into P register. The machine is now ready for the next minor cycle. k of these minor cycles constitute a major cycles, where k is the length of the pattern.

After a major cycle, the content of D(1)...D(s) is the post-cursor-position-bit-vector for the string in X(1)...X(s) in the beginning of the major cycle.

Thm. If n>>s, then b(1)...b(n) can be obtained in at most n/s major cycles with the inputs t(1)...t(n) and p(1)...p(k).

Sketch of Proof. t(1)...t(s)...t(s+k) are taken in the first major cycle to produce b(1)...b(s). t(s)...t(2s+k) are taken to produce b(s+1)...b(2s) at the second major cycle, and so on. It is easy to convince oneself that at most n/s major cycles are needed.

Clearly, to produce b(r)...b(r+s-1), for some r such that $1 \leq r < s$, the inputs t(r)...t(r+s-1)t(r+s)...t(r+s+k-2) and p(1)...p(k) are needed. If t(r+s)...t(r+s+k-2) do not exist, i.e. the subject string is too short, it is necessary to take NONE character (character which does not match any character) as t(r+s)...t(r+s+k-2). As an example, if the subject string is t(1)...t(m), where $k \leq m < s$, the initial loading of t's into X registers would

leave X(n) empty for m<n≤s. It is thus necessary to load X(n) for m<n≤s with NONE characters; furthermore, after each minor cycle, a NONE character has to be loaded into X(s) also.

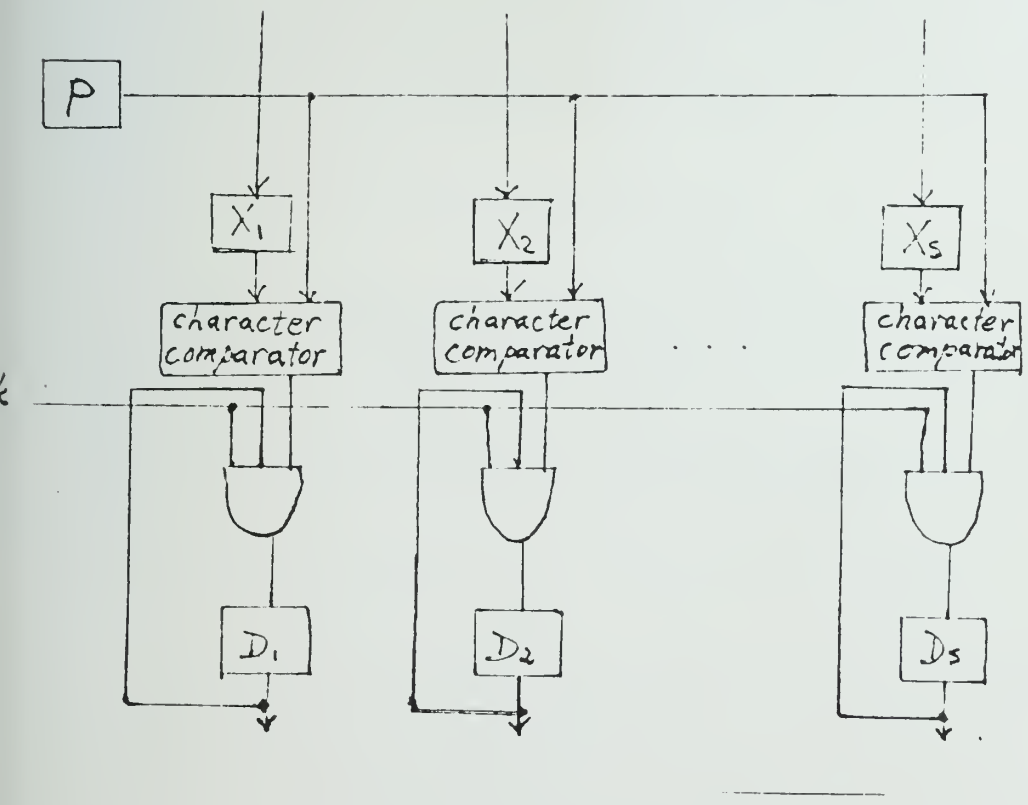


Fig. 3.4.2-1 A pattern matcher

Algorithm 3.4.2-1

```
load D's with 1's (set D's);
r:=1
repeat
   load X(1)...X(s) with t(r)...t(r+s-1). If not enough
   t's to fill all the X registers, pad on the right with
   NONE characters;
   j:=1;
   repeat
      load P(j);
      compare, AND the result with the old D(i);
      store new binary results into D(i);
      shift content of X's left one character position;
      if no more subject string awaiting to
         be processed, stuff a NONE character
         into X(s)
      else
         X(s) receives the next character t;
      j:=j+1;
   until j>length of pattern;
   read out D's. The bit vector formed by
   the content of D(1)...D(s) is a post-cursor-
   position-bit-vector for t(r)...t(r+s-1);
   set D's;
   r:=r+s;
until length of pattern>length of remaining subject string
 t(r)...
```

The working of the PM can best be illustrated by an example. Let us use the subject string abcdefgh and the pattern def. Fig. 3.4.2-2a is a snapshot of the content of the registers before and after the first character comparisons. After the first minor cycle, the contents of X's are shifted left one character position, the new character e of the subject string is loaded into X(s), and the next character of the pattern, e, is also loaded into register P (Fig.3.4.2-2b). The routine of comparison and updating continues.

Fig. 3.4.2-2d marks the beginning of the second major cycle since the content of D represents the post-cursor-position-bit-vector for the substring abcdef. The bit vector in D, which is 0001, indicates that there is a match at position 4 of the subject string.

In Fig. 3.4.2-2d, the contents of X registers are shifted left k-1 positions (where k is the length of the pattern), with successive character of the remaining subject string entering X(s) as it is vacated. Matching begins for this new section of the subject string as it

did in Fig.3.4.2-2a.

The shift which takes place after Fig.3.4.2-2d introduces a null character into X(s), as no more characters are available from the subject string. The whole process terminates at Fig.3.4.2-2f, when the length of the string in X registers is less than k. The complete bit string obtained is 00010000.

content of X (X(1)...X(s)):	a b c d
content of P	d d d d
content of D (D(1)...D(s)) before	1 1 1 1
content of D after	0 0 0 1

Fig. 3.4.2-2a

content of X	b c d e
content of P	e e e e
content of D before	0 0 0 1
content of D after	0 0 0 1

Fig. 3.4.2-2b

content of X	c d e f
content of P	f f f f
content of D before	0 0 0 1
content of D after	0 0 0 1

Fig. 3.4.2-2c

content of X	e f g h
content of P	d d d d
content of D before	1 1 1 1
content of D after	0 0 0 0

Fig. 3.4.2-2d

content of X	f g h λ	
content of P	e e e e	
content of D before	0 0 0 0	
content of D after	0 0 0 0	λ =NONE

Fig. 3.4.2-2e

content of X	g h λ λ
content of P	f f f f
content of D before	0 0 0 0
content of D after	0 0 0 0

Fig. 3.4.2-2f

3.4.2.1 Speed

Using Schottky TTL, each level of AND-gate delay is approximately 10nsecs. Each minor cycle is thus about 70 nsecs.

3.4.3 Stage II

Denote the output of each PM of Stage I by M(T,Ai), where T is the text or subject string and Ai is the pattern.

The function of StageII is to broadcast the output vector of each M(T,Ai) to n places and the output of each M(T,Bj) to m places.

3.4.4 Stage III

It is necessary to shift each vector M(T,Ai) to be ANDed with M(T,Bj) in StageIV |Bj|-|Ai| places to the right (or to the left if |Bj|-|Ai|<0) to ensure that vector-ANDing in Stage IV results in vector containing a

bit '1' wherever Ai concatenated with Bj matches the subject string T. Denote the shifted result of this stage by S(M(T,Ai)).

3.4.5 Stage IV

Vector-AND S(M(T,Ai)) with M(T,Bj) for all combinations of i and j, where $1 \leq i \leq m$, and $1 \leq j \leq n$.

3.4.6 Stage V

Vector-OR the results of Stage IV. The m*n vectors of stage IV are reduced to one vector. The output is the post-cursor-position-bit-vector of the pattern match of the subject string T and pattern

(A1|...|Am) (B1|...|Bn).

3.4.7 Stage VIa

This stage converts the binary vector into a vector of indices, using recurrences similar to the 1's position count of Chen and Kuck [CHE75].

3.4.8 Stage VIb

This is the first-match recognizer which consists simply of a tree of OR gates, which produces a 1 if any of the inputs is a 1. If this output is 1 and the PMN is operating at first-match-only mode, it implies the presence of a match in this major cycle, and the task of the PMN is considered completed without further processing.

3.5 Gate, Time, and Gate*Time

An estimate of the number of gates, the number of gate-delays, and the product of the two are given in Fig.3.5-1 through Fig.3.5-6 for Stage I of Method 1 and Method 2, as a function of text length and pattern length. The text length denotes the length of the subject string the processor can accomodate at one time--it is likened to the word length of an arithmetic processor.

Fig. 3.5-7 through Fig.3.5-12 show estimates of total gates, delay, and the product of the two for Stages

I through Stages VI of Method 1 and Method 2. The pattern assumed is of the form

$$(A_1|\ldots|A_n)\ (B_1|\ldots|B_n)$$

and $|A_1|=\ldots|B_n|=k$.

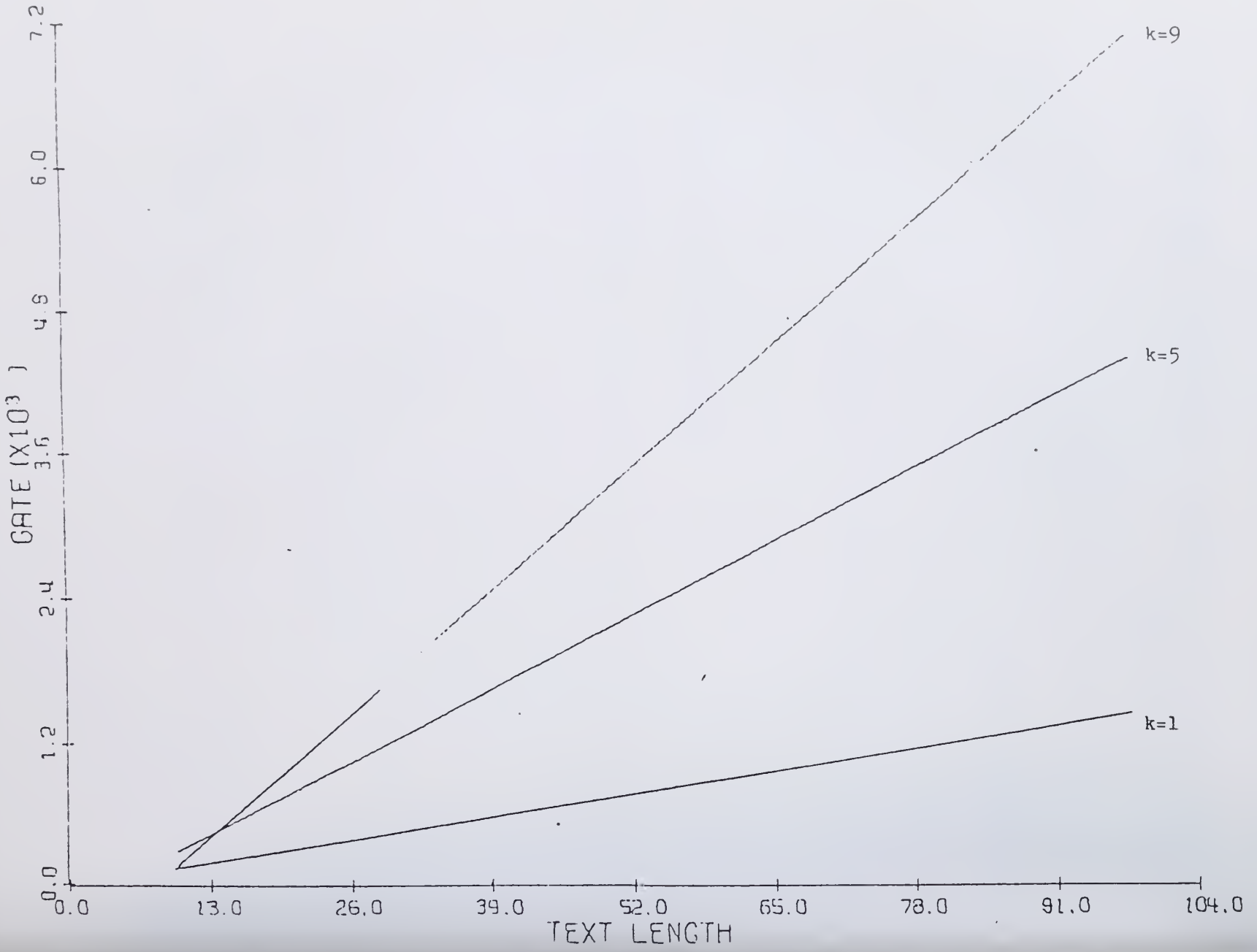


Fig. 3.5-1 Gates for Method 1 Stage I

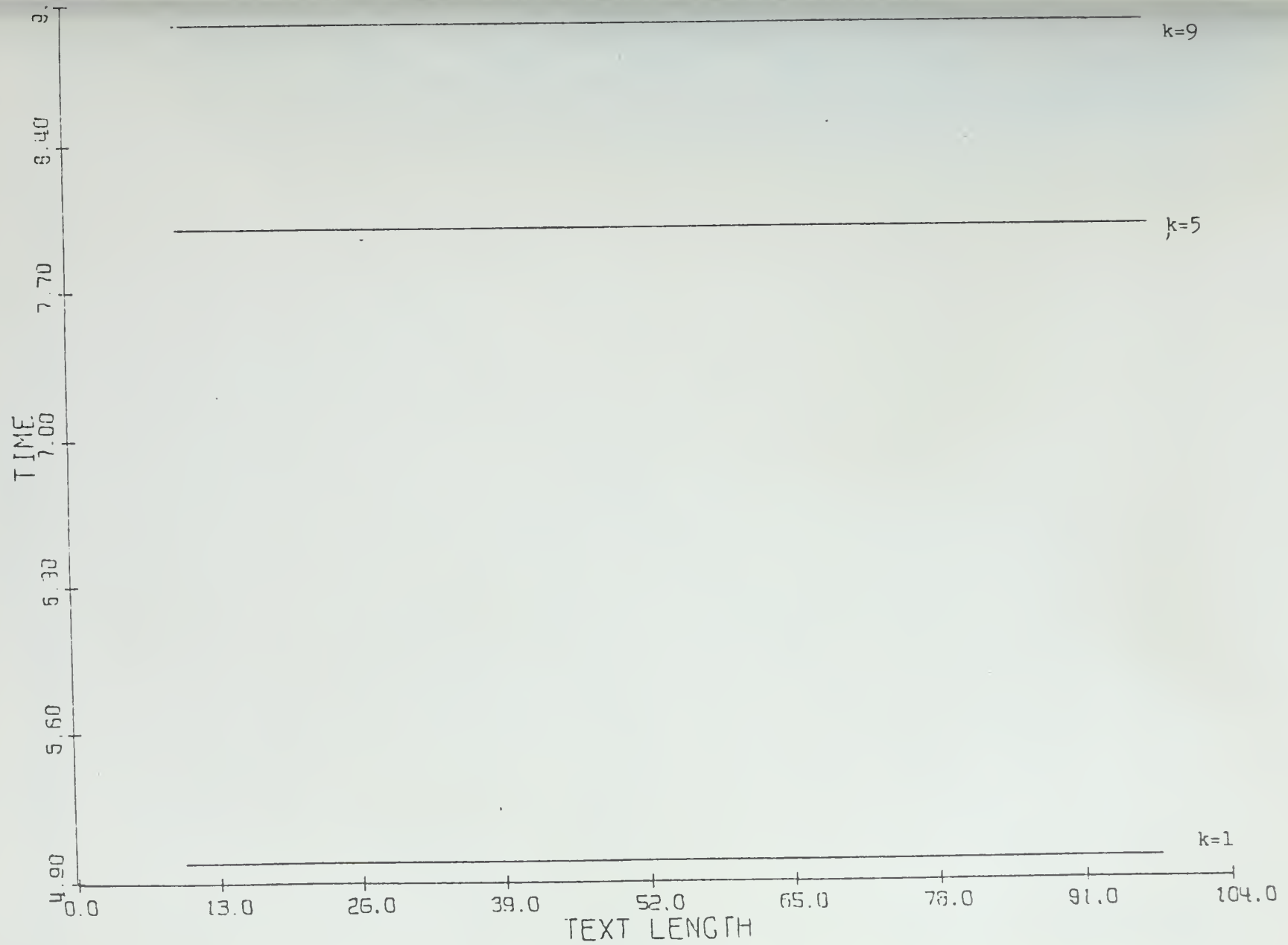


Fig. 3.5-2 Time for Method 1, Stage I

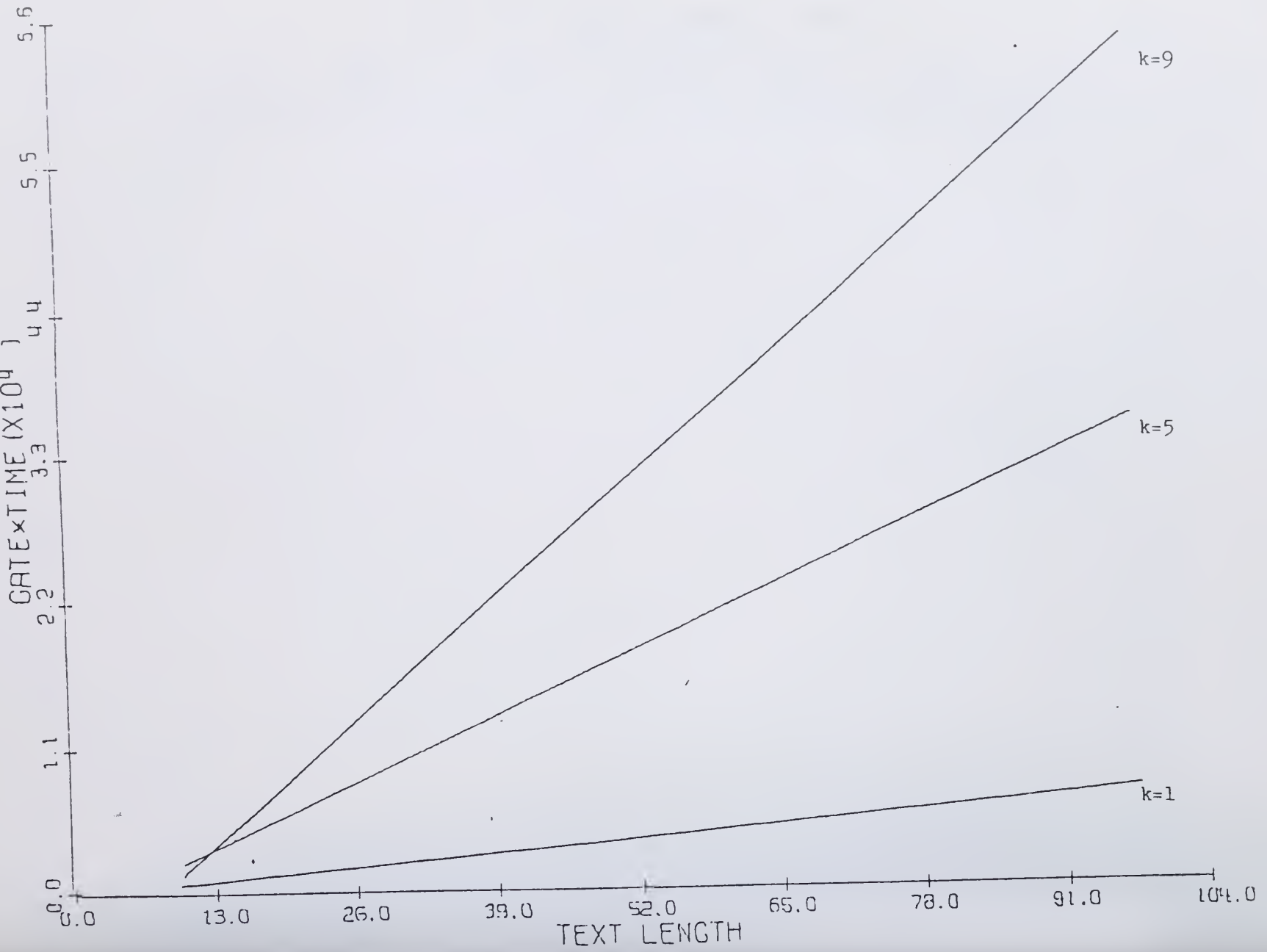


Fig. 3.5-3 Gate*Time for Method 1 Stage I

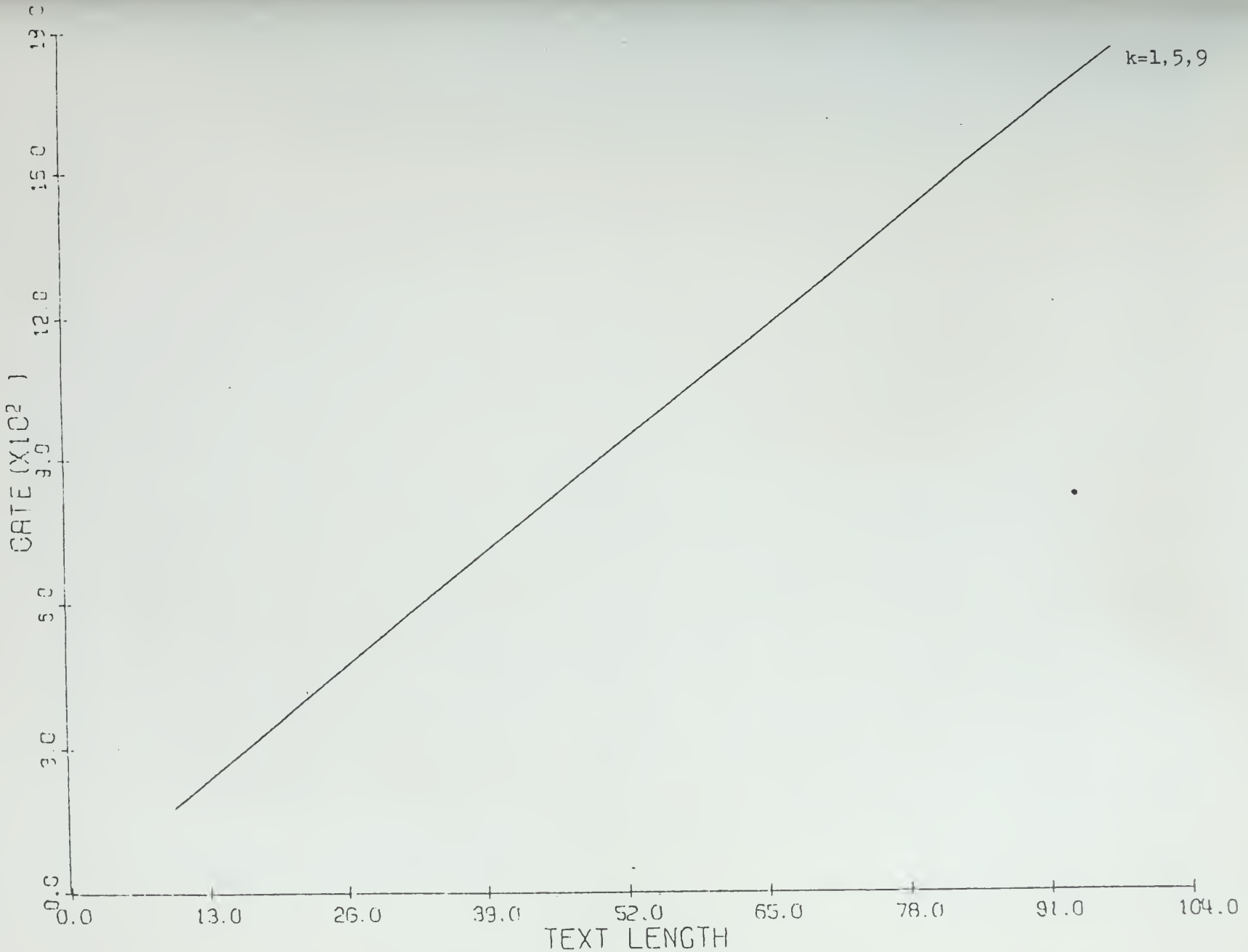


Fig. 3.5-b Gate for Method 2, Stage I

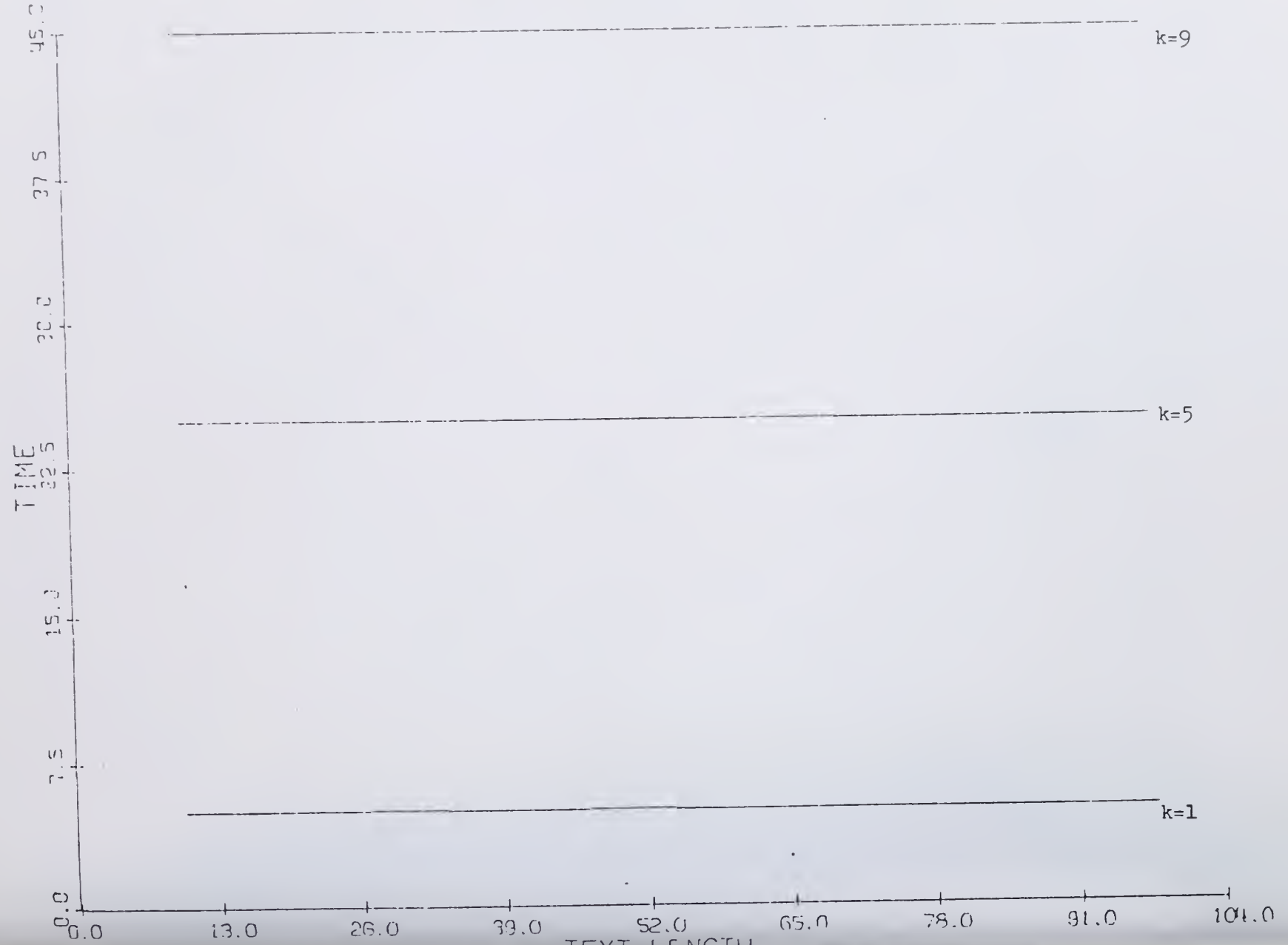


FIG. 3.3-3 Time for Method 2, Stage 1

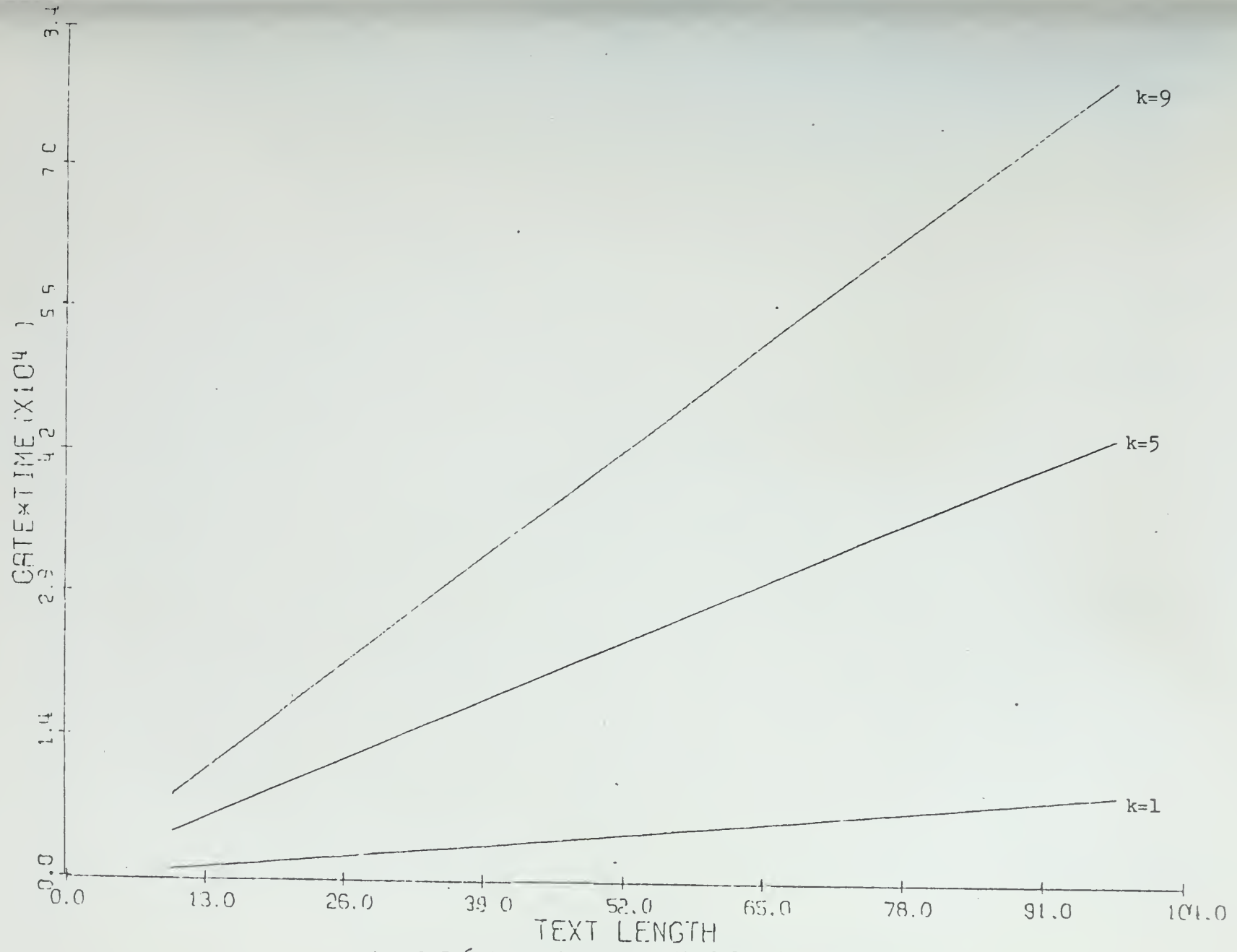


Fig. 3.5-6 Gate*Time for Method 2, Stage I

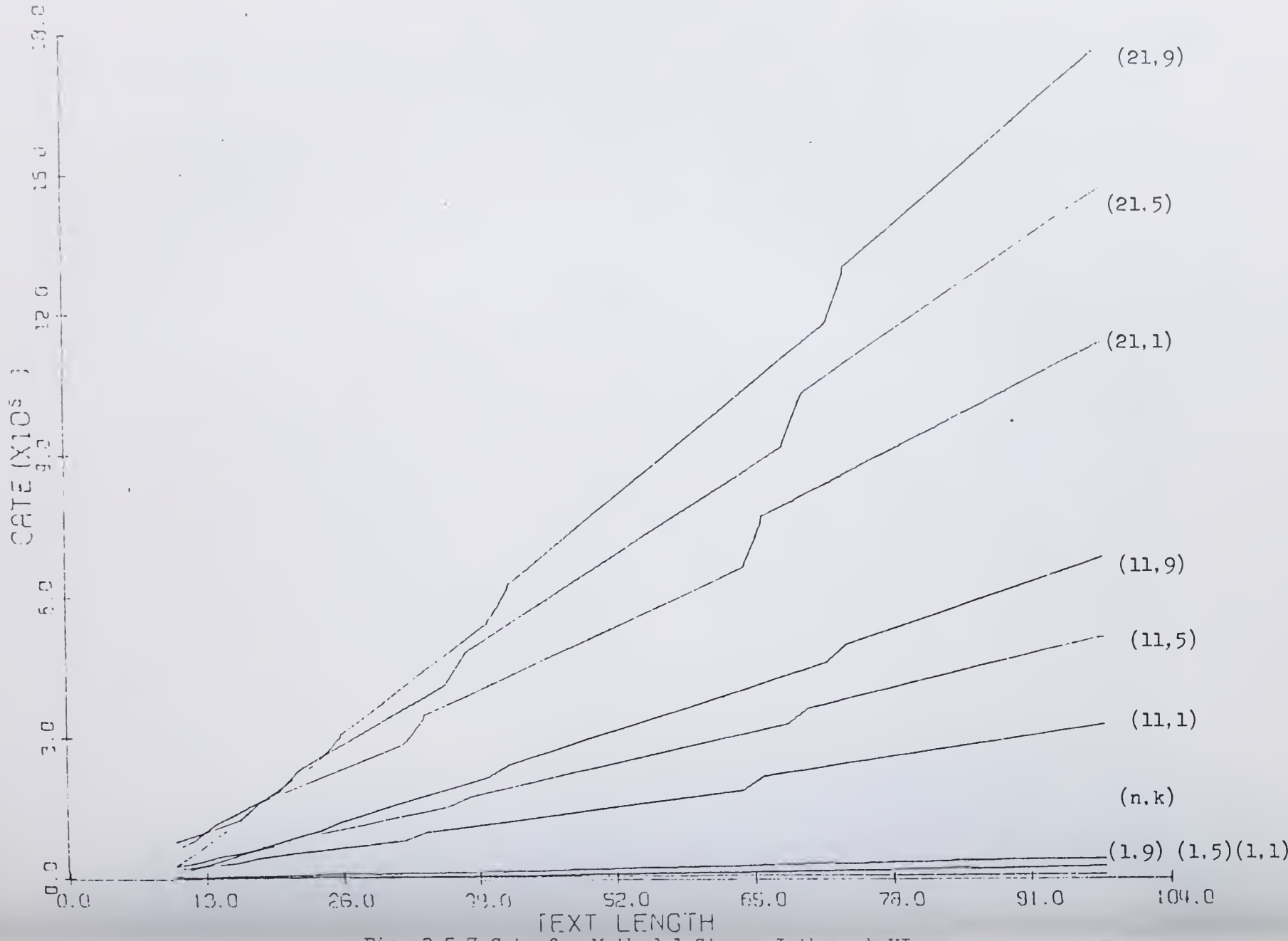


Fig. 3.5-7 Gates for Method 1, Stages I through VI

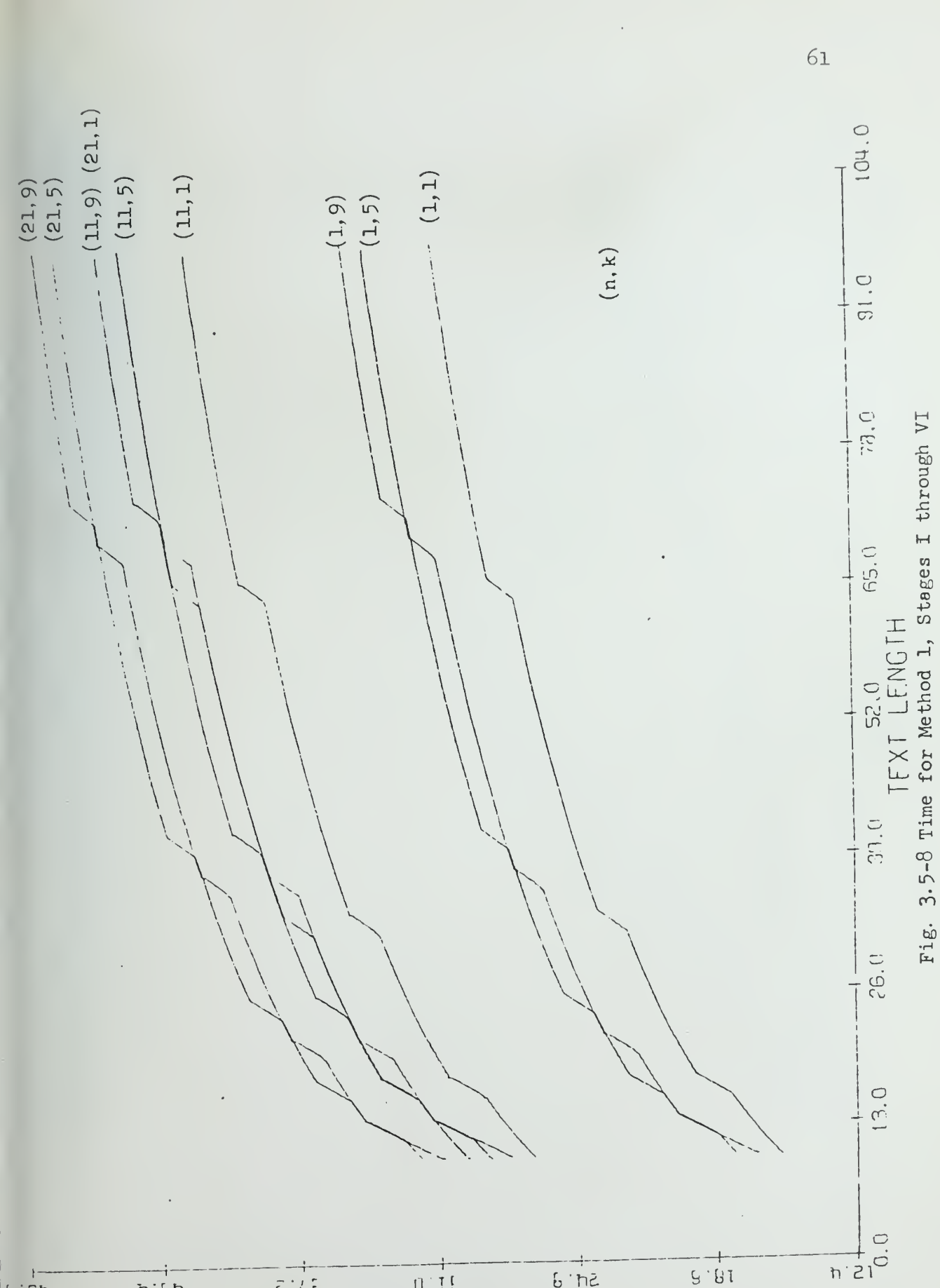


Fig. 3.5-8 Time for Method 1, Stages I through VI

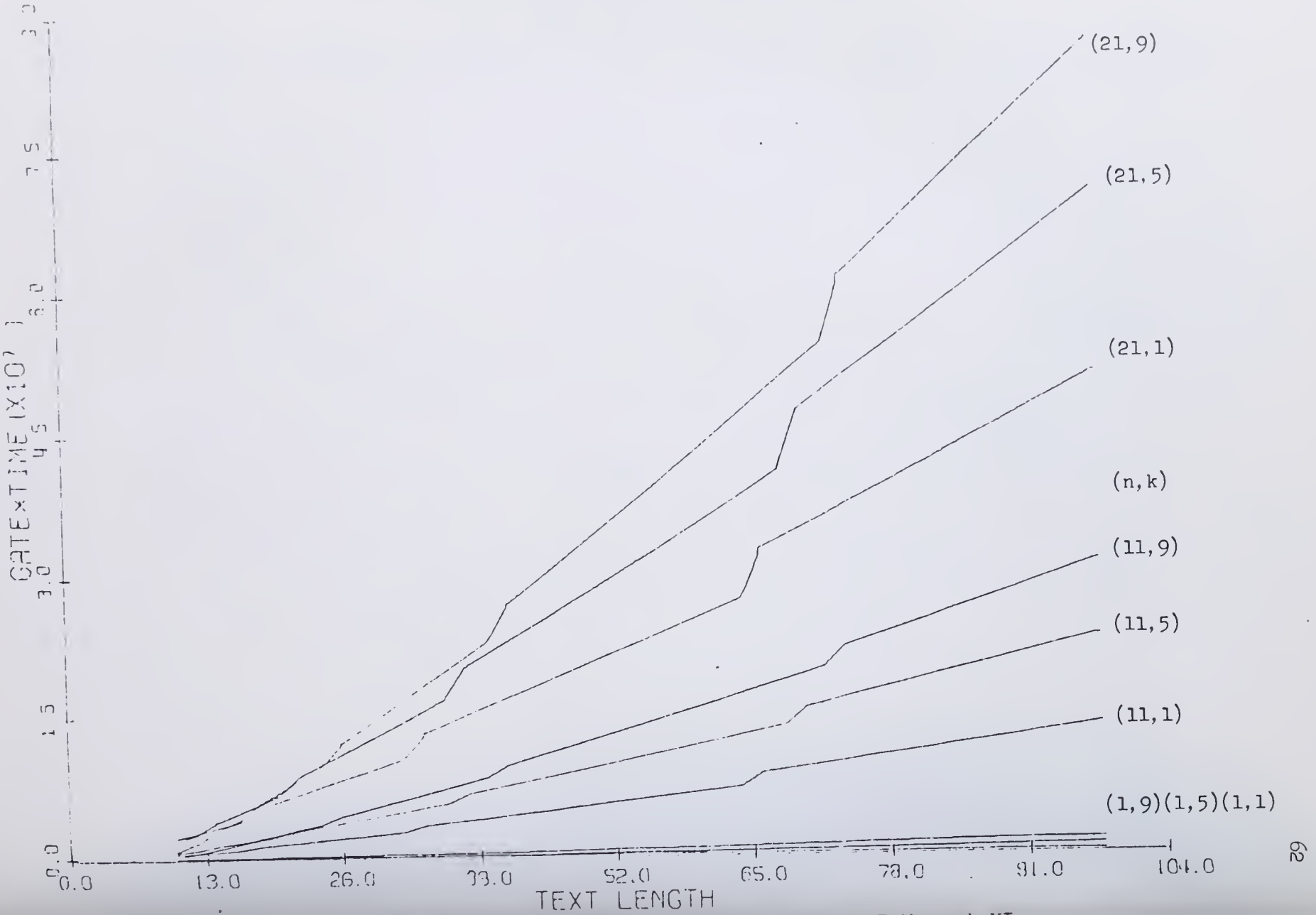


Fig. 3.5-9 Gate*Time for Method 1, Stages I through VI

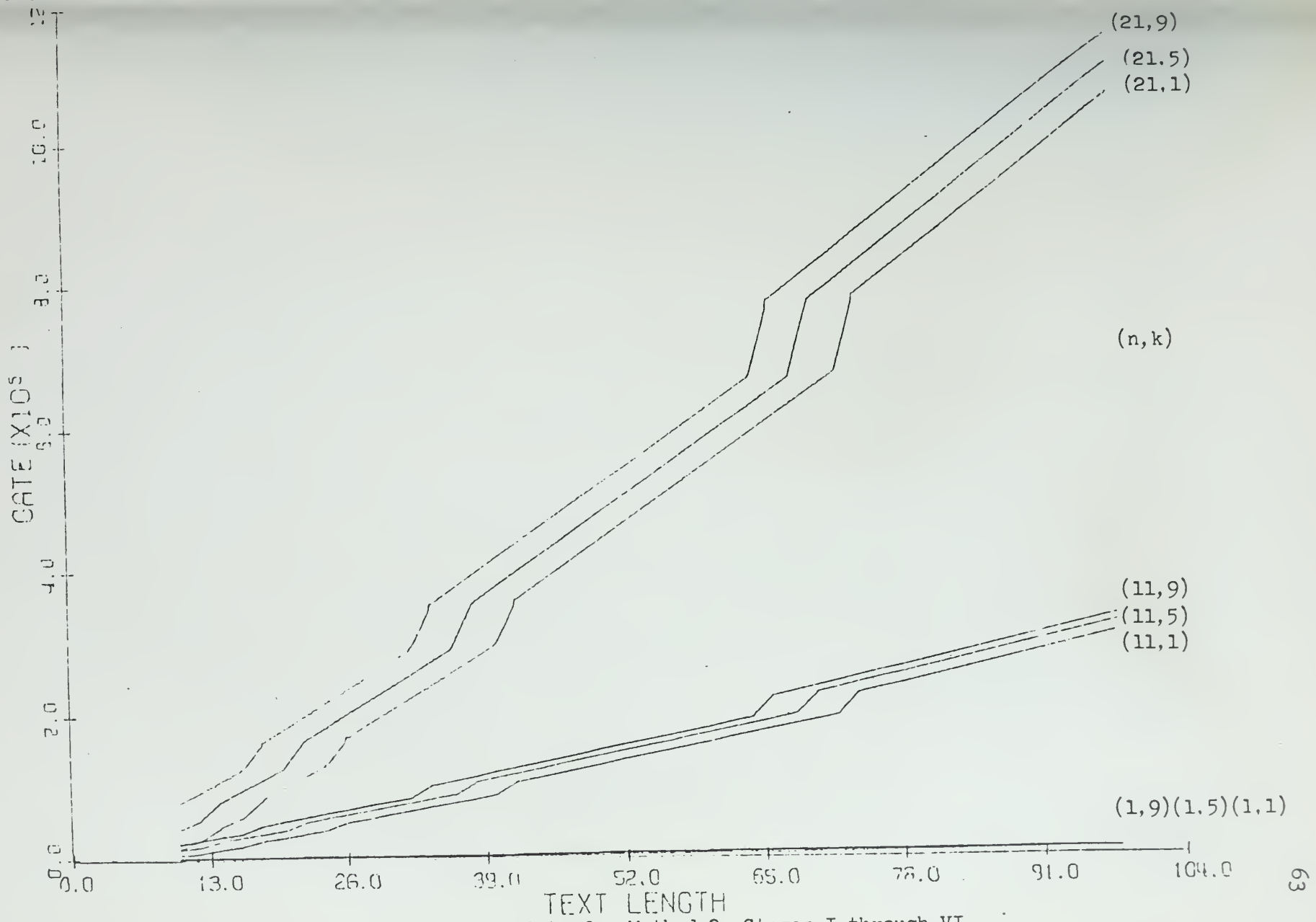


Fig. 3.5-10 Gates for Method 2, Stages I through VI

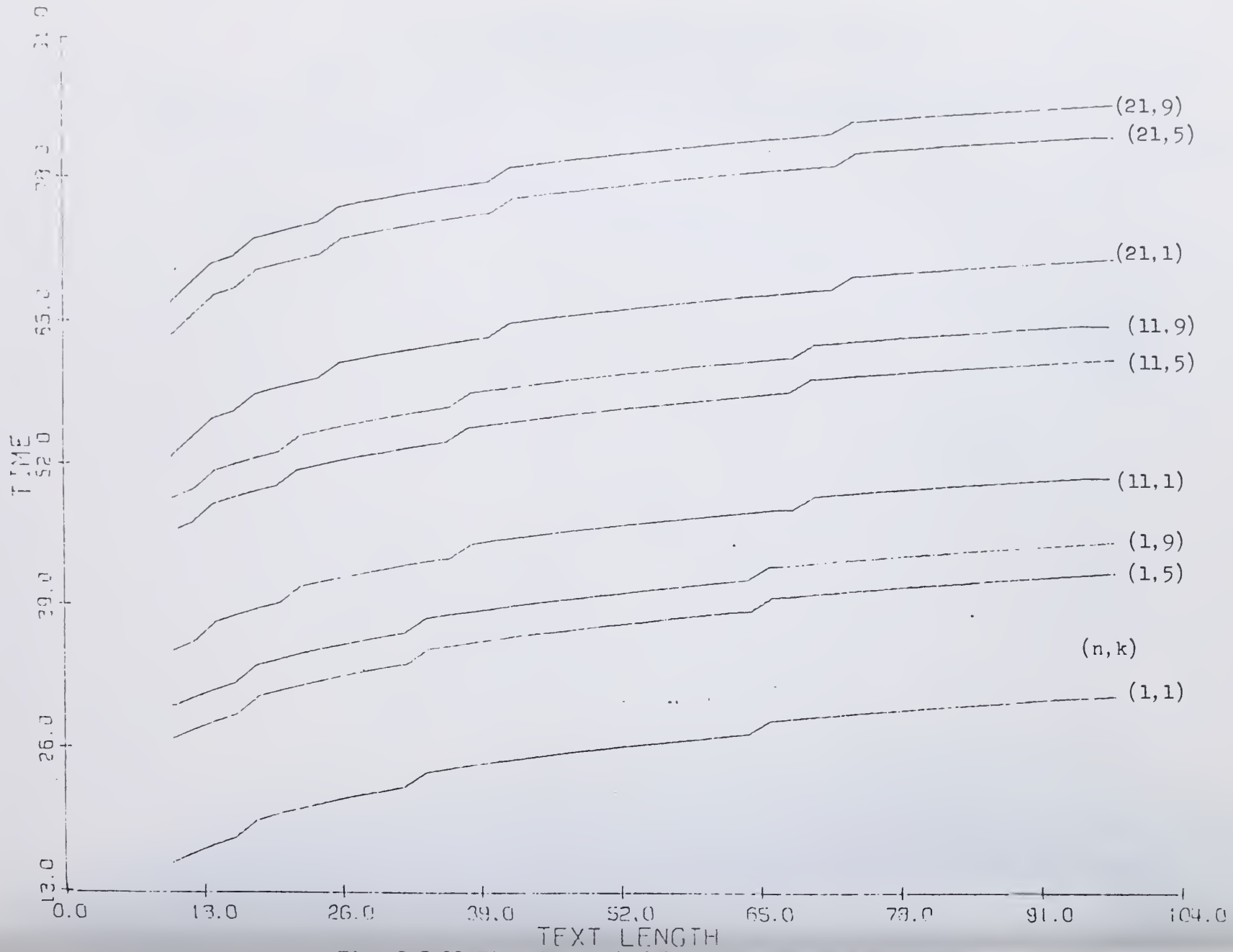


Fig. 3.5-11 Time for Method 2. Stages I through VI

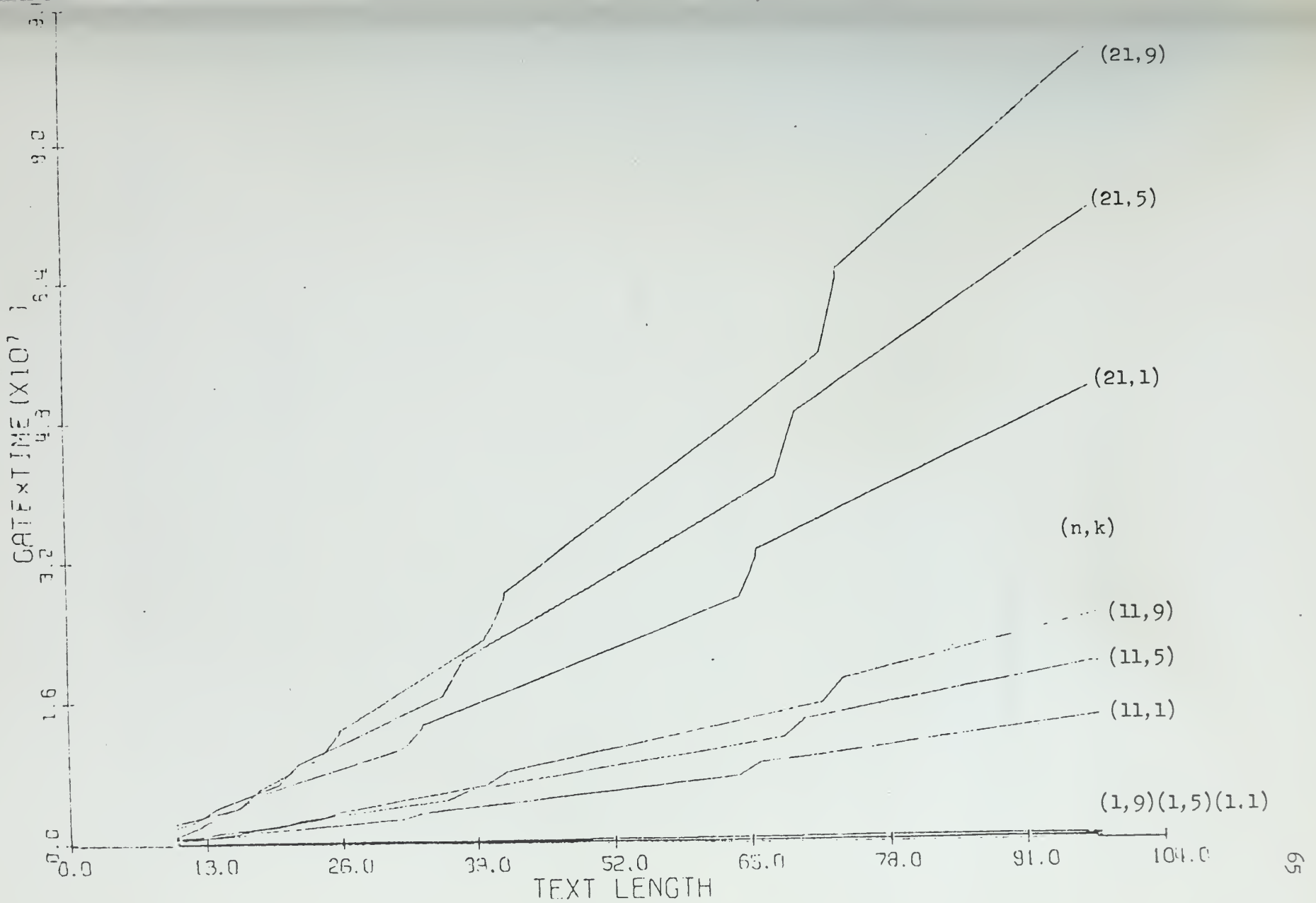


Fig. 3.5-12 Gate*Time for Method 2, Stages I through VI

4. MACHINE DESIGN

4.1 Overall Organization

Based on the characteristics of SNOBOL programs, we have come up with the following list of items which a machine should have in order to speed-up string processing.

1. Since memory reclamation is one of the most frequent activities, it is desirable that the memory unit has a memory reclamation processor (garbage collector). This garbage collector, synchronized with the other memory accesses to avoid races [SIE75], would mark, relocate, update, and reclaim unused memory spaces continually, independently of, and in parallel with other processors except for synchronization.
2. Pattern matching network which would speed-up most of the pattern matching processes.
3. Program instructions and data should be kept separate to abate access conflict.
4. Either a type conversion unit to convert between

different types of data representation or an ALU which performs arithmetic in BCD.

5. Multi-function arithmetic logic unit.

6. Build into the memory units some facilities for the transfer of data within the memory.

7. Control unit which is capable of synchronizing all the parallel activities and avoid incorrect sequencing.

8. IF tree processor [DAV72b] to evaluate decision trees. Results of program analysis shows that IF trees of considerable length exists, that most of the trees are sparse, and that 83% of the IF trees analyzed are less than six levels deep. A decision processor of six level capability would thus be sufficient. Trees greater than six levels could be solved by mapping into free nodes (folding) or repeated use of the processor (iteration).

4.2 Garbage Collection

SNOBOL is a "typeless" language, meaning that it is declaration-free and that storage is not preallocated for

variables, but rather, it is allocated on demand. When storage is no longer needed, it is freed automatically by a process called garbage collection, or memory reclamation. Storage is no longer needed when, for example, a variable is assigned a new string value; consequently, the storage for the old string is no longer in use assuming that it is not referred to by anything else.

Allocation of memory in SNOBOL is very simple. Memory is divided into two regions: allocated and free regions. When storage is needed, it is taken from the beginning of the free region.

The basic garbage collection algorithm [GRI72] works as follows:

1. Mark--identify storage area which is no longer needed (or inaccessible).
2. Relocate--compact the accessible cells into a contiguous region.
3. Update--pointer references to relocated cells have to be updated.

Garbage collection is a time consuming process. Typically, a program spends about 30% of its execution time in garbage collection[GIM/6,STE/5]. Through the use of parallelism, we propose to overlap garbage collection with actual execution of the SNOBOL program.

The garbage collector in our proposed machine is essentially a processor which executes the three phases: (1) mark, (2)relocate, and (3) update, in parallel with the other processors. The most important consideration of this processor is its synchronization with other processors, and a similar problem for the case of LISP[MCC62] has been discussed in some detail by Steele[STE/5].

Note that in addition to reclaiming unused space, continuous garbage collection has the advantage of improving program locality in a paged memory system.

4.3 <u>If Tree Processor</u>

SNOBOL programs tend to have a large number of

decision statements compared to numerical programs like FORTRAN. In order to speedup the execution of decision statements, an IF tree processor [DAV72a] is used. The IF tree processor receives the conditional result set as input, whereby each bit in the conditional result set represents the result of a decision statement. The output of the processor is the identification of the actual path traversed due to the results of all these decision statements. By executing all the decision statements in parallel, passing all these results to the IF tree processor and then choosing the correct results depending on the path, one is able to speed-up the execution of decision statements which otherwise have to be processed sequentially. The compiler algorithm for the IF tree processor is sketched in Section 2.4.4.

4.4 Memory

Before discussing storage requirements and design, it is necessary to consider briefly the internal data representation of SNOBOL.

The basic unit of data representation in SNOBOL is a descriptor, which is a uniform representation of all SNOBOL values and locations. The descriptor can be thought of as the basic 'word' of the SNOBOL system. Every value descriptor contains a type code to indicate the datatype and the associated value. Values with limited field length, such as numbers, are directly represented in the descriptor. Values with unlimited length, for example strings, are represented indirectly by a pointer to a memory location and perhaps an offset from that location. Consequently, memory must be addressable byte-wise and descriptor-wise, just as in some computers, locations can be addressed both as a byte and as a word, where a word is a multiple of bytes. Since the size of a descriptor is far bigger than a byte, a descriptor is thus a multiple of bytes.

Memory is divided into two distinct units: program memory and data memory in order to enhance overlapping in memory access. The program memory stores the code generated by compiling a SNOBOL source program. The data memory holds constants and natural variables generated

during the compiling and execution phases.

4.5 Program Memory

The program memory consists of a hierarchy of storage devices [KUC/0,MAT/2]--the cache memory, primary program memory, and external storage such as CCD,magnetic bubble memory, disks, or drums.

4.6 Data Memory

The data memory is made up of a number of memory banks. As transfer within the memory (stemming from concatenation, garbage collection, etc.) is important in SNOBOL, special considerations are given to the design to speedup 'move' of a sequence of data, from one location to another.

The general organization of a single memory bank is shown in Fig.1.6-1. The smallest addressable entity is a byte, and the largest addressable entity is a descriptor.

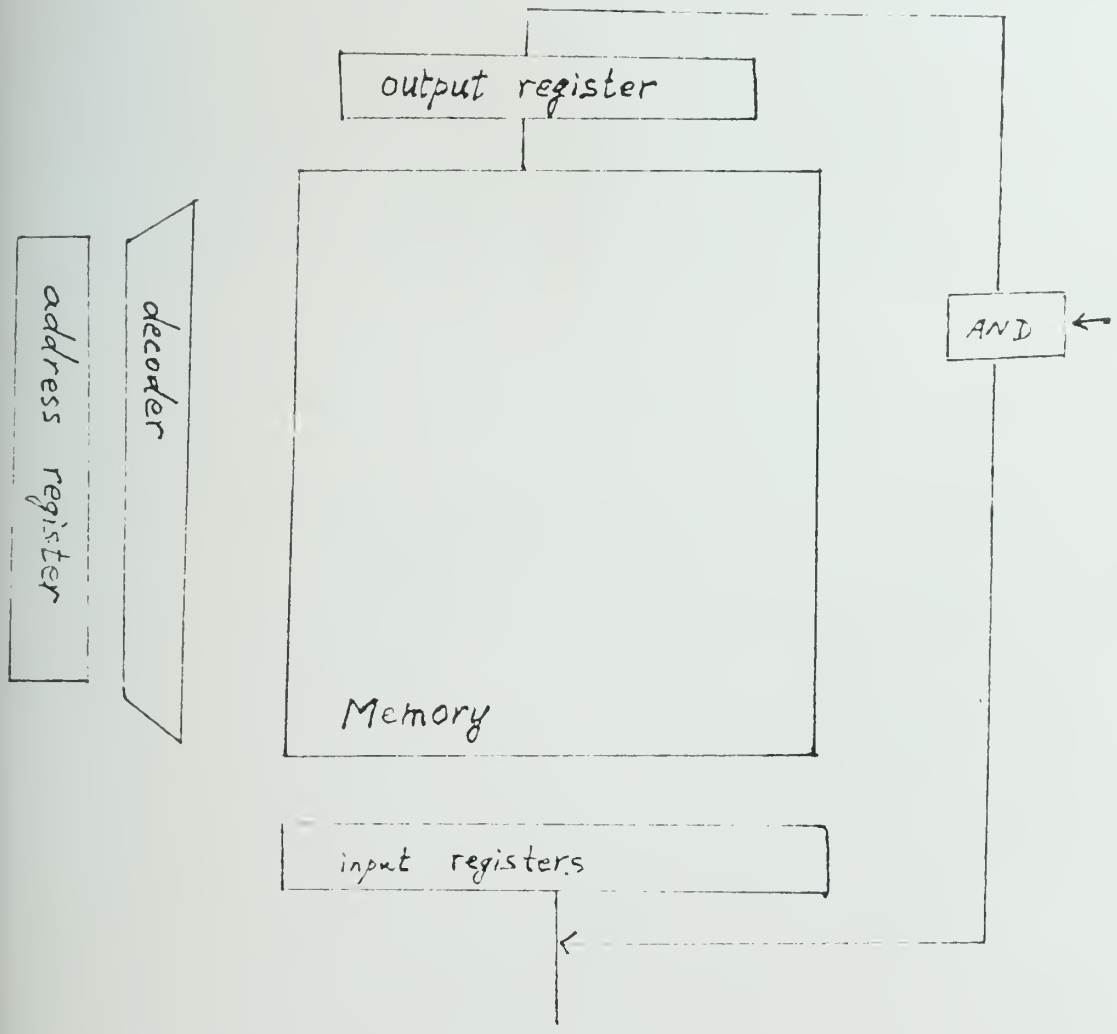


Fig. 4.6-1 A memory bank

For a transfer within a bank, data are read out to the output register, gated into the input register, and a write cycle is initiated. Transfer between memory banks may be performed via the fast inter-memory bus. The inter-memory bus, Fig.4.6-2, is a simple barrel shifter since transfer of a string from one memory bank to another within a multi-banks memory system can be regarded as shifts when looking at the memory ports, as shown in Fig.4.6-3.

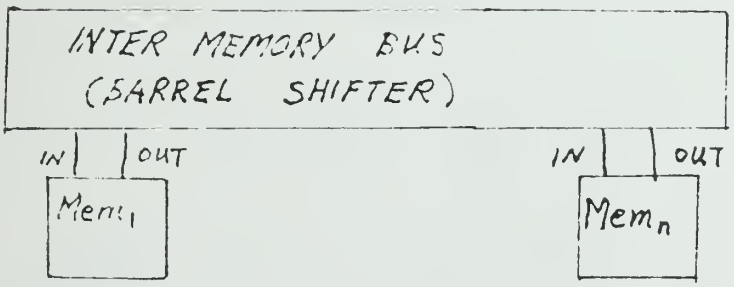


Fig.4.6-2 Inter Memory Bus

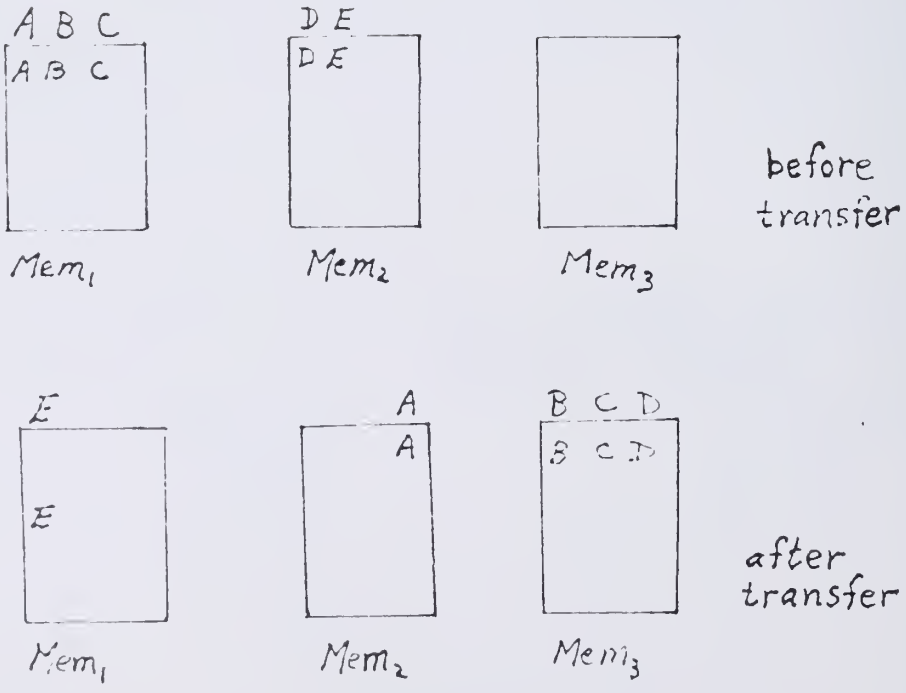


Fig. 4.6-3 The effect of moving a string

4./ Instruction Set

Speed is by no means the only concern which remains for string processing languages; implementation of such a language, related software, and compilation of programs are also far more complicated on a conventional machine. In ALGOL, a statement of the form

A=B+C

can easily be compiled into

```
LOAD B
ADD C
STORE A
```

whereas in SNOBOL, no similar sequence of codes can be generated in the machine level for a string operation such as pattern matching. The gap between the software and hardware has to be bridged by sophisticated software. With the proposed machine organization, we can have an instruction set which includes the one mentioned below.

The instruction set of the machine are given in Table 4.1-1. An address usually designates a processor register or memory location.

In addition to the instructions frequently found in a conventional machine, it has some instructions for pattern constructions and pattern matching.

ARITHMETIC OPERATIONS	
ADD X1,X2	X2:=X2+X1
SUB X1,X2	X2:=X2-X1
MPY X1,X2	X2:=X2*X1
DIV X1,X2	X2:=X2/X1
INC X1	X1:=X1+1
DEC X1	X1:=X1-1
NEG X1	X1:=-X1
NUM X1	convert X1 to a number
AND X1,X2	X2:=X1 X2
OR X1,X2	X2:=X1 X2
XOR X1,X2	X2:=X1 + X2
NOT X1	complement X1
TRANSFER OF CONTROL	
GOTO X	unconditional branch
GOTO X,C	branch to x if the condition specified in C (EQ,NE,LT,LE,GT,GE) matches the condition specified in the condition code register
DATA TRANSFER	
MOV X1,X2	transfer value in X1 to X2
COMPARISON	
COMPN X1,X2	convert X1 and X2 to numbers compare them, and store the result of the comparison (LT,EQ,GT) in the condition code register
COMPS X1,X2	convert X1 and X2 to strings, compare them lexically, and set the condition code register.

Table 4.7-1 Instruction set (continued)

TSTNULL X1	test if the value in X1 is null and set the condition code to EQ or NE
PATTERN CONSTRUCTION	
ALTERNATE X1,X2	return in X2 the alternation of X1 and X2
CONCATENATE X1,X2	return in X2 concatenation of X1 to X2
	immediately assigns the protion of the subject string matched by X1 to X2
CONDITIONAL-ASSIGNMENT X1,X2	same as IMMEDIATE-ASSIGNMENT except that the assignment occurs only if the pattern match is successful
CURSOR X1	assign cursor position to the location in X1
PATTERN MATCHING	
PATTERN-MATCH S,P	match S against P
MATCH-AND-REPLACE S,P,O	match S against P and replace the matched substring in S by O. O may be null.

Table 4.7-1 Instruction Set

5. EMPIRICAL RESULTS

5.1 Introduction

A sensible way to speed-up or design a machine is to study the characteristics of the program that are to be run on that machine. To design a SNOBOL machine, it is therefore desirable to study some of the statistics of typical SNOBOL programs. Of course, there is no such thing as typical program or typical programmer; therefore, we cannot trust any measurement to be very accurate. Nevertheless, it would give us an indication of what these typical machine parameters are.

Static and dynamic statistics are given in this section. Dynamic statistics are generally more useful for computer designers, although they are more difficult to get. Dynamic statistics usually include relative frequency of usage of different operations, and the amount of time different operations take. When the relative frequency of usages of different operations is known, one would know which operations the speedup attempt should be

emphasized on. When the speed of different operations are known, one will have a better idea on how to balance the computer system.

5.2 Static Frequency Count

The results of the analyses of a sample of SIMSCRIPT programs are presented in this section. These programs were unprotected files stored on disks at the Computer Services Office of the University of Illinois at Urbana-Champaign. This sample of 45 programs, written by a wide variety of people, includes a good cross section of application programs; some examples of which are those used for text or file processing, job control language processing, artificial intelligence research, data entry, language preprocessor, etc. The total number of statements in the programs, and the program size distributions are given in Table 5.2-1 and Table 5.2-2 respectively. From these distributions, one can perhaps obtain an estimate of the size of program memory needed.

The static frequency count of different types of

operators are summarized in Table 5.2-3. One would intuitively expect this result that arithmetic operators would appear less frequently than the string operators.

Frequency count of different keywords are shown in Table 5.2-4. ANCHOR, which controls whether pattern matching is to be performed in anchored mode is the most frequently occurred keyword.

According to our analyses, 593 out of the 2520 statements contain decision goto's. This result implies that 23% of the statements are decision statement. The occurrences of different type of 'goto' is illustrated in Table 5.2-5.

The number of levels in each IF tree [DAV72b] is also examined. Table 5.2-6 lists the result of the study. The average number of levels is approximately five, and the median is four. One further observation of the IF trees is that the IF trees are generally very sparse. These data pertaining to IF trees would prove useful to the design of IF processor.

Total number of executable statements	2520
Total number of comment statements	536
Total number of statements	3056
Total number of programs examined	45
Total number of labels	597
Total number of statements longer than 80 chars	78
Total number of statements with complex pattern	45

Table 5.2-1 Characteristic of program examined

Number of statements (excluding comment)	Frequency
1- 50	32
51-100	6
101-150	3
151-200	2
201-250	1
251-300	0
301-350	1

Table 5.2-2 Program size

Type of operators	percent of all operators
concatenation	10.6%
alternation	3.9%
pattern matching	10.0%
assignment (=)	60.8%
arithmetic (+,-,*,LE,...)	10.3% (99% involves +,-)
'$','.'	4.2%

Table 5.2-3 Operators frequency

Keywords	Percent of all keywords
ANCHOR	70%
TRIM	23%
DUMP	2%
TRACE	2%
FULLSCAN	2%

Table 5.2-4 Keywords count

Type of goto	Frequency count
:( )	274
:S( )	305
:F( )	332

Table 5.2-5 Goto frequency

Height (number of levels)	Frequency
1	20
2	15
3	12
4	7
5	3
6	1
7	0
8	1
9	1
10	1
11	0
12	1
13	1
14	3
15	0
16	1
17	1
18	1
.	.
.	.
22	1
.	.
.	.
34	1

Table 5.2-6 Height of IF trees

Length of string constant	Percent
1- 5	69%
6-10	13%
11-15	4%
16-20	3%
21-25	2%
26-	9%

Table 5.2-7

5.3 Dynamic Statistics

The dynamic statistics referred to in this thesis are those of Gimpel [GIM76]. These results are summarized here in Table 5.3-1. The original measurement [GIM76] was done on SITBOL, which is an interpreter. The values in Table 5.3-1 has been obtained from [GIM76] after renormalizing to mask off the effect of the interpreter, and grouped into the five categories shown in Table 5.3-1.

Type of operation	percent of total time spent in program
garbage collection	31.5%
string processing	25.2%
pattern matching	16.1%
I/O	11.2%
others	16.0%

Table 5.3-1 Execution time distribution

6. SYSTEM PARAMETERS

6.1 Introduction

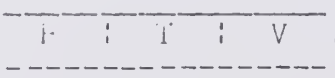
The following sections will describe the system parameters of the machine described in Chapter 4.

6.2 Descriptor Size

Descriptor, as implemented in SNOBOL on the IBM 360, occupies a double word (64 bits). The CDC 6000 descriptor is one word (60 bits). We choose a descriptor size of 64 bits since it seems to be the smallest descritor size that is a power of two and also adequate for its purpose, as manifested by the implementations of SNOBOL for IBM 360 and CDC 6000.

6.3 Descriptor format

All data objects in SNOBOL are represented by descriptor. Descriptor in our machine have the form



The F field is a flag which identify the type of the descriptor The F field contains 8 bits.

The T field represents the size of the object in the V field or pointed to by the V field. Twenty-four bits are allocated to the T field, allowing it to indicate length of string of up to 2**24 bytes long. The 32 bits allocated to the V field allows it to represent integers in the range -2**31 to (2**31)-1. Real numbers may have the approximate range of 10**-78 to 10**75. Pointers (address), when used in this field, may have the range of 0 to (2**23)-1.

6.4 Character Size

We propose that our machine use 8-bit characters since it is conveniently a power of two and it provides a large enough character set that is useful in text processing. An 8-bit character will also provide a character set which includes a special character which matches any character and a character which does not match any character.

6.3 ALU Processor

The ALU processor is a multifunction processor with one unit for division, one for multiplication, and six units of adder since 99% of the arithmetic (such as addition, subtraction, test for greater than, equality, etc.) require the adder.

Among the programs analyzed, there are only five statements which require multiplication or division; thus, only one of each is used in our machine. Six adders are chosen for the sake of convenience (so that the total number of functional units add up to eight). A more accurate reflection of the number of adders needed can be obtained by transforming the programs and counting the number of additions and subtractions that can be carried out in parallel.

Several logic processors (AND and OR) are used in the pattern matching network.

6.6 IF Tree Processor

Statistics from program anslysis indicates that approximately 25% of the statements in a program are decision statements, which agrees very well with the other languages, such as COBOL[STR74] and GPSS[DAV74b]. The average number of levels for the decision tree is five, and that the tree is very sparse; thus, a decision processor that can handle a six level decision tree should be sufficient.

With a six level decision processor,it can handle at least six decision statements at once. In view of this grouping of six decision statements into one, the percentage of decision statements in a program may be regarded as 4%. Consequently, one IF tree processor for our machine is sufficient.

6.7 Program Memory

As can be seen in Table 5.2-2, SHOBOL programs are generally small in size. To obtain an estimate of the

number of words needed for the program memory, we assume (1) most programs are less than 350 statements long, (2) a safe estimate of 10 words (64 bits) of machine instructions are needed for each statement. Using the above assumption, a program memory of 4K words will be adequate.

6.9 Data Memory

The number of data memory banks required is determined by the memory bandwidth required. The PMU requires 9 new characters from the memory per minor cycle (70nsecs), one character for the text and eight for the patterns; consequently, the necessary bandwidth is 9 characters per 70 nsecs.

Assuming one microsecond memory is used, a simple calculation shows that sixteen banks of memory, each with a descriptor size of 64 bits (8 characters) are needed to fulfill the memory bandwidth requirement.

6.10 PMN

Based on the observation of pattern matching in SNOBOL, we propose a PMN that can take patterns of the form

pattern = (A1¦A2¦A3¦A4) (B1¦B2¦B3¦B4)

where Ai and Bj are strings. The connection can be changed to handle patterns in other form. A PMN of this complexity contains eight pattern matchers, and can also be used as eight pattern matchers by intercepting the output of Stage I.

The PMN also contains 64 AND-processors, each of which can AND two 64 bit vectors, producing a 64 bit vector as a result. In addition, the PMN also has 32 OR-processors.

The parameter s (section 3.4.2) of the pattern matcher is chosen to be 128.

6.11 Expected Speedup

With the autonomous garbage collector, garbage collection time is expected to approach zero. Concatenation, in SPITBOL, requires 0.05+0.0005n milliseconds on an IBM 360, where n is the number of characters involved. In our system, only one microsecond is needed to move 128 bytes; thus, the concatenation time of our system is approximately n/128 microseconds. The speedup (time for SPITBOL/time for our system) is about 640 for large n.

Pattern matching time is the sum of the time for subject evaluation, pattern building, scanning, and object evaluation and replacement. The replacement operation is approximately equivalent in time to two concatenations, and we have already seen the speedup achievable.

In SPITBOL, pattern matching for

```
P = DUPL('A',100)
S ('A' 'B') 'C'
```

requires 12 milliseconds plus overhead. Using our machine, the time is approximately 30 nanoseconds (20 gate delays * 10 nanoseconds per gate). The speedup in this case is 1.2*10**6.

7. CONCLUSION

We have pointed out some architectural requirements for a language like SNOBOL, proposed an organization for the machine, and suggested some transformations and compiler algorithms that can be used on SNOBOL. We have also estimated the amount of speedup achievable using the special hardware mentioned earlier.

A few areas are opened up for more research by this project. First of all, more dynamic statistics on SNOBOL are desirable. Secondly, some simulations can be done to determine more accurately the speedup achievable. Simulation and program analysis for FORTRAN has been going on for quite a few years [KUC72].

LIST OF REFERENCES

[AHO75] Aho,A.V. and Corasick, M.J. "Efficient String Matching:An Aid to Bibliographic Search," Comm.ACM, vol.18,6.

[BAC59] Backus,J.W. The Syntax and Semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM Conference," Proc. International Conf. on Information Processing. UNESCO (1959). 125-132.

[CHE75] Chen,S.C. and Kuck,D.J. "Combinational Circuit Synthesis with Time and Component Bounds," Dept. of Comp. Sci. Rep. No. 75-775, Univ of Illinois, Urbana.

[COH65] Cohen,K. and Weastein,J.H. "AXLE: An Axiomatic Language for String Transformations," CACM 8,11.

[DAD65] Dadda,L. "Some Schemes for Parallel Multipliers." Alta Frequenza, 12,5,349-356.

[DAV72a] Davis,E.W. "Concurrent Processing of Conditional Jump Trees," COMPCON 72 Digest of Papers. 279-281.

[DAV72b] Davis, E.W. "A Multiprocessor for Simulation Applications," Ph.D Thesis, Univ. of Illinois at

Urbana-Champaign, Dept. of Comp. Sci. Rep. No. 627.

[DiFo6] di Porino,A.C. "String Processing Languages and Generalized Markov Algorithms," Symbol Manipulation Languages and Techniques. Proc. of the IFIP Working Conference on Symbol Manipulation Languages, D.B.Pobrow (ed.). North Holland Publishing Co.

[FiS71] Fischer,M.J. and Paterson,M.S. "string Matching and and Other Products." TR-41,Project MAC. MIT, 1974.

[GI'73] Gimpel,J. "A Theory of Discrete Patterns and Their Implementation in SNOBOL4." Comm. ACM, 16,91-100.

[GI'76] Gimpel,J.F. Algorithms in SNOBOL4 (John Wiley & Sons, N.Y.).

[GRI77] Griswold,R.E. The Macro Implementation of SNOBOL4. (W.H. Freeman and Company, San Francisco).

[Her64] Herstein,I.N. Topics in Algebra (Xerox College Publishing).

[KNU71] Knuth,D. "An Empirical Study of FORTRAN Programs." Software 1,2,105-133.

[KNU74] Knuth,D.E., Morris,J.H.,Jr., and Pratt,V.R. "Fast Pattern Matching in STRINGS," FRCS-74-440,Stanford

University, Stanford, California, 1974.

[KRA72] Kraska,.W. "Parallelism Exploitation and Scheduling," Ph.D Thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science Rep. No.518,1972.

[KUC70] Kuck,D. J. and Lawrie, D.H. "The Use and Performance of Memory Hierarchies: A Survey," Software Engineering, vol. 1, Academic PRESS, 1970,45-77.

[KUC72] Kuck,D.J. Muraoka,Y., and Chen,S.C. "On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and Their Resulting Speedups," IEEE Trans on Computers,21,12,1293-1310.

[MAT72] Mattson,R.L. and Traiger,I.L. "Storage Hierarchy Design," COMPCON 72 Digest of Papers. 145-148.

[MCC62] McCarthy,J., et.al. LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass.

[MSO61] MacSorley,O.L. "High-Speed Arithmetic in Binary Computers," Proc. IRE,42,1,67-91.

[MUR71] Muraoka,Y. "Parallelism Exposure and Exploitation in Programs," Ph.D. Thesis, Univ of Illinois at Urbana-Champaign, Dept. of Computer Science Rep.

p. 421, 1971.

[ADAM] Adam, P. (ed.) Revised Report on the Algorithmic Language ALGOL 60. CACM 5,1,1-17.

[POA/I] Poage, J.R. and Polonsky, I.P. The SNOBOL4 Programming Language (Prentic Hall, N.Y.).

[STE/I] Stewart, J.F. "An Algebraic Model for String Patterns," M.S. Thesis. Dept. of Comp. Sci., 38-39. Univ. of Toronto.

[STR/I] Stroble, J.L., Jr. "Multiprocessing Compactifying Data Collection," Conf ACM 1,2.

[STR/I] Strozendt, W.R. "Program Speedup through Concurrent Record Processing," Ph.D. Thesis, Univ of Illinois at Urbana-Champaign, Dept. of Comp. Sci, Rpt. No. 638.

[TOW/I] Towle, R. "Control and Data Dependence for Program Transformations," Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science Rpt. No. 76-788.

[WAI/I] Waite, W.M. Computer Programming with GPPL II. If Press.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-77-907	2.	3. Recipient's Accession No.

4. Title and Subtitle

Multiprocessor for String Manipulation

5. Report Date: October, 1977

6.

7. Author(s): Wing Kai Cheng

8. Performing Organization Rept. No. UIUCDCS-R-77-907

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.: US NSF MCS73-07980

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered: Master's Thesis

14.

15. Supplementary Notes

16. Abstracts

A machine organization is proposed for the fast processing of SNOBOL programs. Measurements of SNOBOL programs are given to support various choices. Since pattern matching is a major time consumer in SNOBOL, special hardware is proposed for it. Several alternative designs were studied in detail and cost-effectiveness measures are given for these.

17. Key Words and Document Analysis. 17a. Descriptors

High-level language architecture
Pattern matching processor
SNOBOL machine
SNOBOL measurements

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

Release Unlimited

19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 106
20. Security Class (This Page UNCLASSIFIED	22. Price

BOUND BY

UNIVERSITY OF ILLINOIS-URBANA
3 0112 000447992